EuroPLoP 2008: 13th Annual European Conference on Pattern Languages of Programming

Irsee, Germany, July 9-13, 2008.

Edited by
Till Schümmer *, till.schuemmer@fernuni-hagen.de
Allan Kelly **, allan@allankelly.net

* FernUniversität in Hagen
** Software Strategy Ltd.

Table of Contents

1. Collaboration & Management

Workshop Leader: **Allan Kelly**

2. Engineering

Workshop Leader: **Dietmar Schütz**

1. Bitstream X-Coder Dietmar Schütz
2. Modular Hot Spots: A Pattern Language for Developing High-Level Framework Reuse Interfaces using Aspects André L. Santos, Kai Koskimies
3. Patterns for Managing Data in Complex Automatic Identification and Data Capturing Environments Diethelm Bienhaus
4. Intelligent Subject - Adapting Observer with push model and filters to handle divergent update needs Paul G. Austrem
5. Advanced Synchronization Patterns for Process-Driven and Service-Oriented Architectures Carsten Hentrich, Uwe Zdun

3. Humans and Engineering

Workshop Leader: **Klaus Marquardt**

1. Developing GUI Applications: Architectural Patterns Revisited Alexandros Karagkasidis
2. Patterns for Licensing Web Services G.R. Gangadharan, Michael Weiss, Vincenzo D'Andrea
3. Using a Profiler Efficiently Tim Wellhausen
4. Patterns for Robust and Flexible Multimodal Interaction Andreas Ratzka

4. Pedagogy

Workshop Leader: **Christian Kohls**

1. Turning me on, turning me off Christian Kohls, Tobias Windbrake
2. Patterns for Supervising Thesis Projects Axel Schmolitzky, Till Schümmer
3. Activating Students in Introductory Mathematics Tutorials Christine Bescherer, Christian Spannagel, Wolfgang Müller
4. Didactic Design Pattern "Highlights" Sven Wippermann
5. Guess my X and other Techno-pedagogical Patterns Yishay Mor

5. Systematic Approaches

Workshop Leader: **Uwe Zdun**

1. Choose Your Own Architecture: Interactive Pattern Storytelling James Siddle
2. Patterns and their Impact on System Concerns Michael Weiss
3. Experiences in Using Patterns to Support Process Experts in Wizard Creation Birgit Zimmermann, Christoph Rensing, Ralf Steinmetz
4. Modeling Architectural Pattern Variants Ahmad Waqas Kamal, Paris Avgeriou, Uwe Zdun

Focus Group Reports

Junkies Like Us: How the Social Web Influences Our Understanding Of Privacy Andreas Rüping

A Collective Social Learning Pattern

Valerie A. Brown,
Director, Local Sustainability Project
Fenner School of Environment and Society
Australian National University

EuroPLoP Workshop, Klosters Irsee, Bavaria
July 9-13, 2008

Abstract

Human-caused changes to the planet have led to worldwide social and environmental disruption. Many of the changes so produced are wicked problems: that is, problems which lie outside the current capacity of the society to resolve them. Resolving such problems requires comprehensive social change. This in turn calls for collaboration among the multiple knowledges into which Western thinking has become divided: individual, community, specialised, organisational and holistic ways of thinking. The global dominance of Western specialised knowledge acts as a barrier to the collective social learning involving multiple knowledges needed for significant change. It also blocks non-Western countries from accessing their own local knowledge.

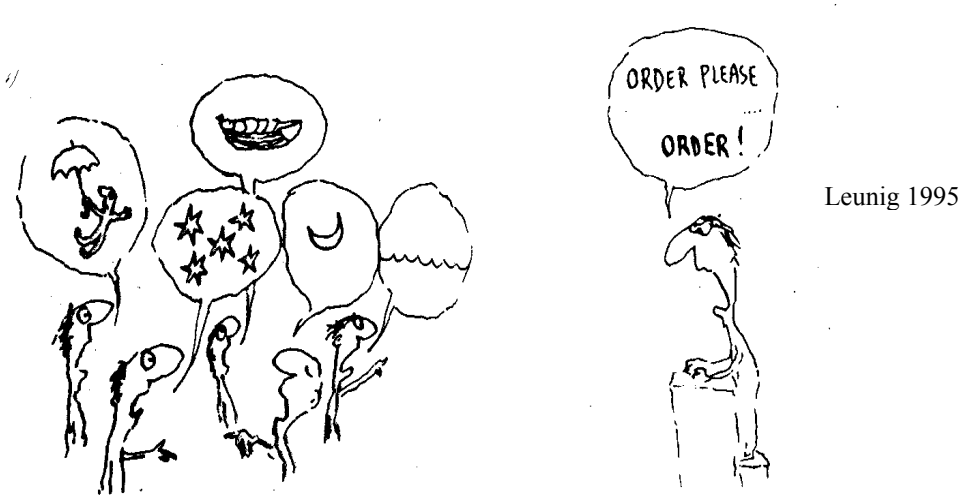
The collective social learning pattern seeks to re-align the multiple knowledges in a way that allows for collective thinking and collaborative practice. A meta-pattern takes the form of a spiral of active intervention that brings the knowledges together on equal terms at each of the four stages of the social learning cycle. Each of the knowledges is a pattern in itself. Each of the learning stage requires an integrative process linking the patterns. As a coordinating framework, the collective social learning pattern combines the multiple knowledges in answering each of the questions in turn: *What should be?* (sharing ideals); *What is?* (establishing facts); *What could be?* (creative ideas); and *What can be?* (collaborative action). The proposed pattern has been trialled in over 200 projects for whole-of-community change. Examples provided here are programs for transformational change in a future-oriented coastal city, an agricultural region, integrated research policy, and a change management workshop.

Keywords: wicked problems, knowledge cultures, social learning, collective thinking

Proceedings of the 13th European Conference on Pattern Languages of Programs (EuroPLoP 2008), edited by Till Schümmer and Allan Kelly, ISSN 1613-0073 <issn-1613-0073.html>.

A collective social learning pattern

Valerie A. Brown



Leunig 1995

A Pattern:
As an element in the world, a pattern is a relationship between a certain context, a certain system of forces which occurs repeatedly in that context, a problem arising from those forces and a social re- which allows these forces to resolve themselves in a collective solution. after Christopher Alexander 2002

1. Context: Societies are being destabilised by the emergence of wicked problems

For many of the issues of the current century, specialised approaches have proved highly successful. The green revolution, extraction of fossil fuels, and giant engineering projects have been achieved through highly specialised, linear inquiry. Multidisciplinary research has been appropriate here. On the reverse side, there has been a chronic inability to bring the specialised disciplines together with the other knowledges necessary for far-sighted decisions on matters that affect the whole of humanity.

Global social and environmental changes and their accompanying local disruption have their origin in the successes of the Western scientific tradition . Yet it is within this same specialised scientific tradition that people continue to look for solutions. The situation meets the definition of a wicked problem, that is, a problem that cannot be resolved from within the thinking of the society that produced it. Horst Rittel in 1973 gives the characteristics of a wicked problem as (with the example of climate change):

> 1. **There is no final solution:** since a wicked problem is part of the social fabric in which it sits, any resolution of the problem leads to social change, and so generates fresh problems that need new solutions.

2. **Every problem is unique**: a complex social-environmental problem can only be understood as the product of a society at a given time and place.

3. **Using existing solutions can impede essential change**: concentration on what works now restricts the capacity to creatively explore *what could be*

4. **Confusion between facts and values:** in complex issues, the distinction between fact (*what is*) and value (*what should be*) becomes confused.

5. Solutions come from unexpected sources: paradoxes are signals of where a society is unstable, and so offer fruitful areas for social learning and change.

The global institutions charged with addressing the planet's wicked problems have been pleading for over two decades for community, specialist, government and industry collaboration as the basis for sustainable solutions (WSSD 2002, Millennium Development Goals 2001, USA Research Institutes 1999, UNEP 1995, WHO 1986, WCED 1986). While Web 2.0 allows open, shared communication among the widest range of individual players, formal organisations and social structures have not been able to follow suit.

2. Problem: The Western division of knowledge blocks resolution of wicked problems

The 'Scientific Enlightenment' of the 17th century has led to our addressing complex problems through a particular problem-solving style. Problem resolution by objective reasoning and reducing issues to their component parts led to semi-miraculous feats, such as eliminating smallpox and placing a man on the moon. On the other hand, the dominance of this way of thinking has blocked the development of other ways of resolving the many wicked problems that cannot be solved through this process.

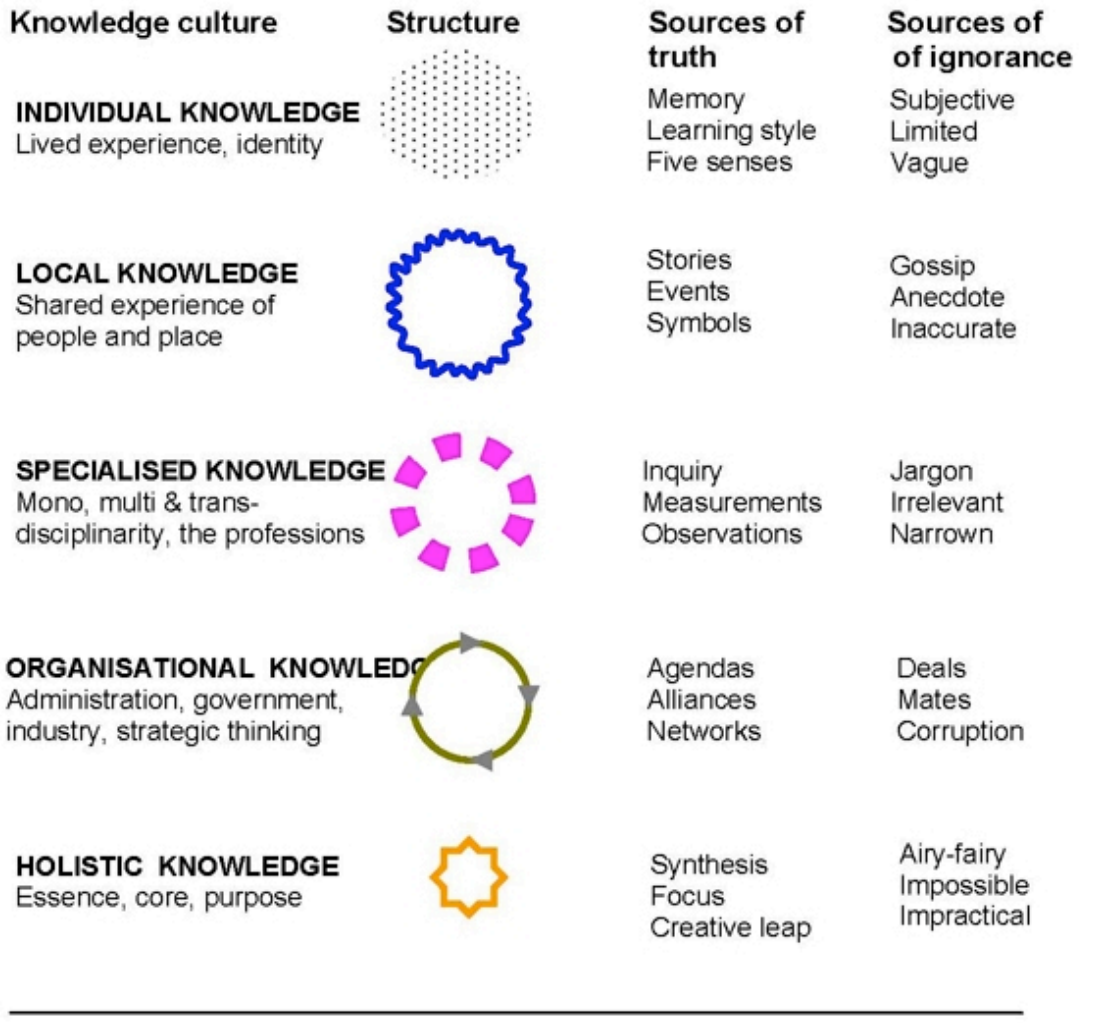
Any complex social-environmental issue provides a case in point. A sustainable resolution of a wicked problem will involve addressing the issue as a whole, thus requiring collaboration among key individuals, affected communities, relevant specialists and influential organisations (both government and industry). Persistent Organic Pollutants (POPs as described in the Wikipedia) are one example. Consider if Environmental Protection Agencies were to become serious about enforcing licensing control of discharges containing POPS into their rivers. Even in small concentrations, POPs are the triggers for a number of cancers and auto-immune diseases. Banning their use would be expected to be a straightforward decision.

There is an obvious benefit from improving the public's health and lowering the cost of the health services. Our water agencies would have lower treatment costs. Communities would avoid the fear of unknown chemicals in their drinking water, pay less rates and have lower individual health costs. A signal would be sent to global corporations that these countries were serious about managing global change, and so stir them to cleaner production. The local natural environment would revert to a self-managing system, achieving efficiencies at no cost to anyone. Surely a win-win-win.

But polluting industries pay rates and provide local employment, so governments are not over-zealous on enforcement. Scientists learn where it is better not to bother placing their research, when there is no uptake of the results. Governments and public health officials are loath to alert communities to risk, citing a greater risk from public panic. Organisations arrive at 'gentleman's agreement' on how much community participation it is politically safe to allow. Even though POPs have blamed for the worldwide 50% reduction in human male sperm there has been no public outcry on such a socially sensitive subject. So the status quo persists, even in the face of the benefits to everyone from a change, and the risks of everyone losing from lack of change.

Figure 1.

The knowledge cultures of Western decision-making (Brown 2001)



A question we continually need to ask is "Who owns the problem?", leading to "Who owns your health?" "Who owns the polluting chemicals?" "Who owns the river?" leading on to "Who owns the planet?" and "Who owns the future?". The questions are answered quite differently by different players, making it hard to achieve

collaboration on sustainable solutions to the collective problem. It proves to be even harder to bring the interests together in the first place.

In a five year collaborative action research program, the Local Sustainability Project (Brown 2008) found significant blocks in the way of collaboration among different ways of knowing were found to be woven into the social fabric. Individuals, communities, specialists, organisations and creative thinkers in all of the 200 communities in the study used different languages to describe the same issue, chose different avenues of action, worked to different action times and were directed towards different outcomes (Figure 1). Such patterns of difference were not primarily matters of right and wrong. They were different interpretations of the same reality, each internally consistent and valid within their own terms. Each produced a different version of reality, isolating each version in a different knowledge culture. An example of a two year program in Townsville, a tropical coastal city is at Appendix 1.

The five year program was able to draw several conclusions. First, each of the contributing knowledges was so self-contained as to form a distinctive knowledge culture, each with its own version of reality. Each had its own internal structure, tests for truth, accepted content and form of language, summarised in Figure 1. So-called collective decisions are usually turned inwards towards integration within each knowledge culture. Rarely are they aimed at the knowledge cultures connecting to each other. As a result, it is difficult for whole-of-community decisions to be achieved in practice (Brown 2008).

The second finding is that, in any collective decision, the familiar trio of community, specialists and organisation are joined by two further knowledge cultures. The personal perspectives of the individuals involved, and the creative contributions of holistic thinkers are rarely recognised as essential contributions to management decisions. From Figure 1 it can be seen that these are knowledge cultures in themselves. Across the field studies, the constructions of reality held by key individuals, and presence of the creative leaps of holistic thinkers, were the variables that allowed connections to be forged among the other three; bookends to the more visible knowledge cultures.

The third finding was the experiential learning process identified by David Kolb forms a thread connecting all of the knowledge cultures (Kolb 1984). One of the significant differences between the cultures is their mode of learning. Individuals learn through their personal experience, communities encase their shared history. Specialists learn through research designs, organizations through strategic planning, and holistic knowledge by free use of the imagination. In every case, however, the learning process follows the steps of Kolb's experiential learning framework (Aslin and Brown 2005). Although given different titles in each culture, the study found that learning always goes through the steps outlined in Figure 2.

The study confirmed the findings of Kolb and his colleagues that, for any given form of management, one of the four stages is given greater emphasis than others (Kolb 1984). Administrators and organizational executives emphasise the reflective, *what should be* stage, while specialists remain focused on observations of *what is.* Successful managers of social change projects and holistic thinkers make the imaginative leap of *what could be*. The skilled professions tend to be concerned with the pragmatic and objective outcomes of *what can be*. So although all learning

follows the same pathway, each different knowledge culture tends to contribute to only one of the learning stages (Kolb, D.A., Lublin, S. and Spoth, J. 1986)..

A fifth finding was that each knowledge culture protected its boundaries by rejecting the others, perpetuating the myths that separate the knowledges, namely that:

- organisational management is self-serving and untrustworthy;
- specialised knowledge is isolated and impractical;
- individual knowledge is biased and limited;
- local knowledge is anecdotal and unreliable; and
- holistic, or focussed knowledge is too difficult to achieve.

In practice, decisions allocated to any one knowledge culture were found to include unacknowledged contributions from each of the others. Every manager is an individual, a community member, has a particular expert training and can think holistically, at the same time. As an individual, all those constructions of knowledge inform every decision, explicitly or implicitly. However, each individual learns to restrict themselves to rely on one aspect of knowledge to the exclusion of the others, as a manager, a specialist or a community member, whichever is their primary concern.

Long-term change depends on including all the knowledge cultures in completing the full learning cycle. Without acknowledgement of these essential aspects of how we build our knowledge base, integrated decision-making in Western thinking can become self-defeating. The knowledge cultures at present form a hierarchy. Organisational and specialized knowledge vie for supremacy, with individual and holistic knowledge largely discounted. Each knowledge culture is likely to addressed only one learning stage. Collective decisions are then unable to complete the learning cycle as a coherent set, and so cannot establish lasting social learning.

In Figure 1 the columns represent the elements which make up each knowledge culture. The rows describe each knowledge's content, power structure, and tests for truth, in turn. These are not the problem. They are the powerful parts of a collective solution. The problem lies in fourth column: the labels with which each knowledge culture rejects and is rejected by the others. The driving forces which maintain the problem lie within the current social system built around the divisions. The solution lies within each of the knowledge cultures. Each needs to develop their own particular contribution to collective learning, rather than continue to use their skills to maintain their existing boundaries.

The phrase 'collective social learning' should be a tautology. Under current social conditions, however, neither standard interactions nor hoped-for collaboration are collective. The knowledge cultures are at best divided, and at worst generate conflict.

Therefore:

- **the solution is to redirect the divided knowledge cultures into a coherent system of collective social learning, in which each knowledge culture is respected and contributes towards a larger whole.**

3. Forces supporting and impeding the solution

Box 1. Contra-example: One-dimensional organisational knowledge:

An award-winning documentary film, *The Corporation* is based on a key 19th-century law that treats companies as persons under law. By bestowing on them the rights and protections that people enjoy, this allows a firm to act as singularly self-interested. Its purpose is solely to create wealth for its shareholders. It puts others at risk to satisfy its profit-maximising goal, harming employees and customers, and damaging the environment. It has no empathy, refuses to accept responsibility for its actions and feels no remorse. In short, the corporation is clinically a psychopath.

Lucy Hughes of Initiative Media, an advertising consultancy, is shown musing about the ethics of designing marketing strategies that exploit the tendency of children to nag parents to buy things, before comforting herself with the thought that she is merely performing her proper role in society. Mark Barry, a 'competitive intelligence professional', disguises himself as a head-hunter to extract information for his corporate clients from rivals, while telling the camera that he would never behave so deceitfully in his private life. Human values and morality survive the onslaught of corporate pathology only via a carefully cultivated schizophrenia: the tobacco boss goes home, hugs his kids and feels a little less bad about spreading cancer.

Company executives and foot soldiers alike will identify instantly with this analysis, because it is accurate. Source: *The Economist* print edition, September, 2004

The fictional anecdote in Box 1 illustrates the ethical chasm between individual and corporation problem-solving. Using the organisational culture as an example, the tale highlights the extent to which each knowledge culture can eventually become inwardly turned, servicing its own internal interests, rather than the needs of its consumers, or the society in general. This has been well-documented for professional fields as far apart as medicine, economics and education. For communities it gives rise to the NIMBY (Not IN My Back Yard) syndrome of local responses to global issues.

The Local Sustainability Project identified the supporting and impeding factors from 30 collective social learning projects that were addressing wicked problems of local sustainability (Brown 2008). Each knowledge culture carried its own impeding and supporting forces into the collective enterprise:

	Impeding forces:	**Supporting forces:**
Individual		
•	individuals alienated from society	reflexive learning by individuals
Community		
•	fragmented and dislocated communities	unifying sense of place
Specialists		
•	monodisciplinary investigations	transdisciplinary scholarship
Organisations (industry and government)		
•	compartmentalised organisations	learning organisations

Integrators
- lack of skills in holistic focus — icons, diagrams, and symbols

Collective thinking
- conflicts of interest — shared social learning

Many of the supposedly integrative responses to complex, dynamic issues maintain the original single-track thinking of each of the knowledge cultures. This means they deal with only one segment of a whole-of-system change at a time. Human activities are far more complex than that. The current tendency to frame any wicked problem in terms of competing interests (say, between local action and government, or between individuals and organisations) simply preserves the current destructive oppositional form of debate.

Standard pattern designs often have the same problem. Each pattern is a valuable exercise in itself. However, one static pattern alone, left without a multi-knowledge framework which recognises inter-connections and dynamic change, serves to trivialise a wicked problem. Patterns that address the complex problems associated with global climate change cannot bring about lasting change unless embedded in an encompassing framework that all participants can share. Such a pattern needs to take account of the knowledge construction of the society that produced them, and the social practices of the society that will implement the solution.

4. Solution: reconnect the divided knowledge cultures in a collective learning spiral

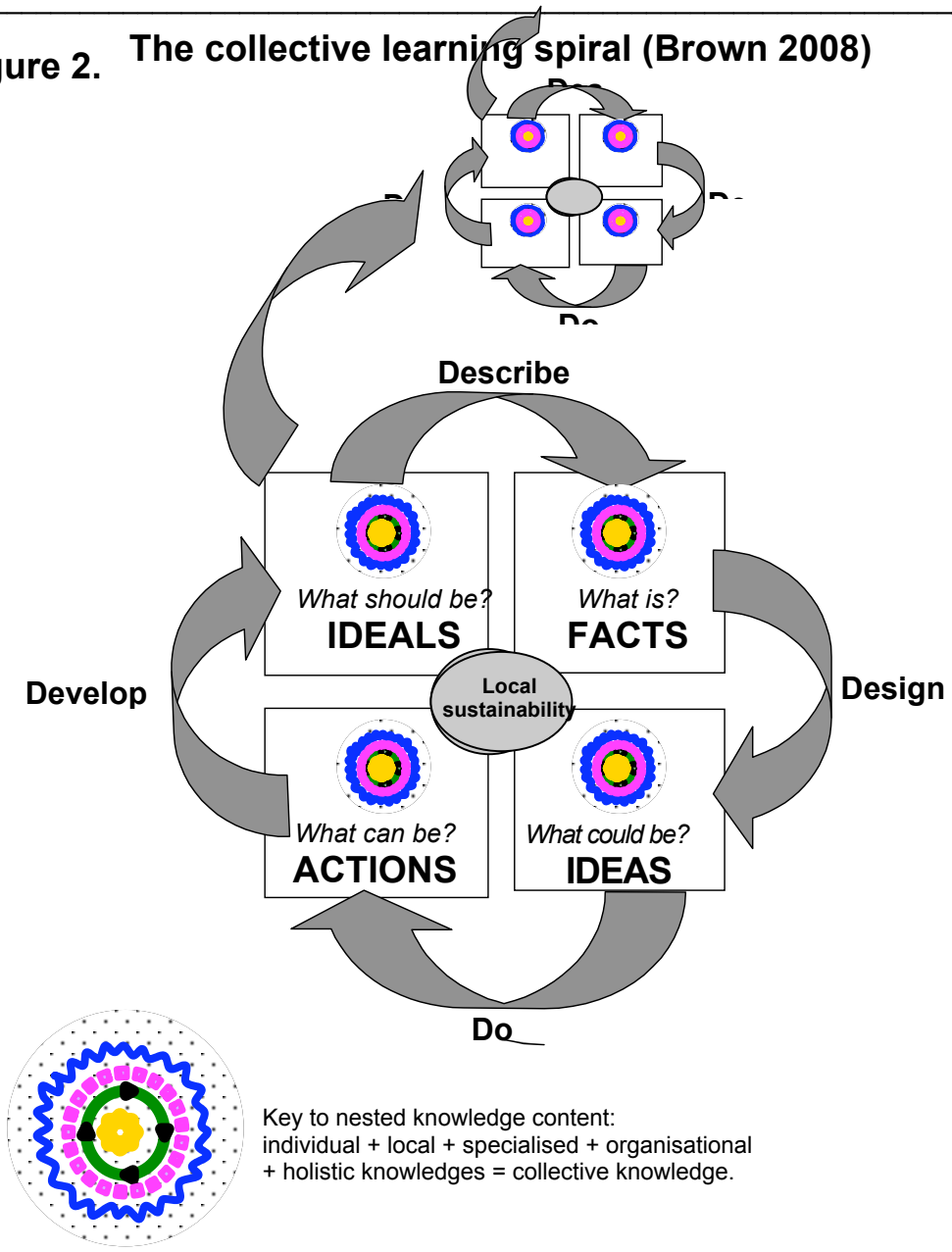
Many integrative frameworks are being developed to address the challenge of re-aligning the divided knowledge cultures. Examples are double and triple loop learning, critical adaptive management systems and resilience thinking. The integrative pattern described here is derived from the human capacity for social learning. It seeks to repair the Western society's fragmented construction of knowledge through a collective learning spiral which incorporates the complexity, dynamism and cumulative nature of whole-of-system change.

Figure 2 offers details of the contributions of each knowledge culture to a collective knowledge culture based on collective social learning. All the Western knowledge cultures are expected to contribute equally to each of the stages of social learning. Continually repeated, the stages form an on-going learning spiral. This process is outlined in some detail in the example of *A Sustainable Townsville* (Attachment 1).

The stages of the collective learning spiral have been derived from Kolb et al, 1984, who undertook decades of observations of effective adult learning. The result was the identification of a repeating cycle of four stages: 1. Developing principles (*what should be?*), 2. Establishing 'the facts' (*what is?*), 3. Brainstorming the potential (*what could be?*) and 4. Putting the result into practice (*what can be?)* (Keen, Brown and Dyball 2005). Unless the full cycle is completed, no long-term learning can occur. Traditionally the learning cycle is followed through separately, by an individual, or a group, an expert or an organisation. Moving to collective thinking and action means

that these different interests are reconciled at each stage. Since human learning is cumulative, in practice the cycle becomes a spiral.

Figure 2. **The collective learning spiral (Brown 2008)**



The same four stages which make up one turn of the collective social learning spiral (Figure 2) can be identified in the change processes for each of the separate knowledge cultures. These are individual experiential learning; community development; action research; strategic planning; and the creative process. The cycle is thus a consistent pattern found in the learning processes of each of the knowledge cultures. Ten in-depth action research studies of transformational social change programs established change programs using the collective learning cycle, therefore meeting Alexander's criteria for a pattern.

5. Examples of the solution in practice

Attachment 1 offers a step by step case study of the way in which the multiple Western knowledge cultures can be brought together within the proposed pattern for collective social learning. The pattern has been repeated many times in the collaborative action research program of the Local Sustainability Project. Here we describe the use of the pattern in developing a behaviour change framework for regional agriculture, a comprehensive rural research policy, and a debriefing process for a workshop team applying the pattern.

Case study 1. Sustainable regional resource management
In the case of a region of exhausted agricultural and natural resources, the focus question was: *How can this region change to support sustainable agriculture?* Those who came together to answer the question were drawn from 10 rural industries, five sub-regions, government agencies, regional opinion leaders, and the coordinating Catchment Committee who funded the study.

What should be?

Seven characteristics of a good life in the region: managing change, having accountability systems, using market mechanisms, working with whole supply chain, establishing collaborations, finding life-work balance, achieving on-ground sustainability, and making the system work for you.

What is?

Each contributing group described a different reality, bringing a deeper understanding of the region's strengths and weaknesses.

What could be?

Change strategies that could satisfy the seven characteristics of a good life in the region:

What can be?

Each industry and region described strategies from their field of interest, providing a powerful overall program of behaviour change.

Case study 2. National rural research program
For future-oriented rural research, the question was: *How can we develop a future rural research policy based on the findings of our past research programs?* This brought together research interests from city and country, government and industry, a wide range of specialists and farmers and graziers.

What should be?

Answered almost unanimously as "through greater collaboration among aqll the members of the policy community".

What is?

This question produced dramatic anecdotes of lack of collaboration and only a few positive examples.

What could be?

The group developed a comprehensive agenda o unrealised opportunities for collaboration.

What can be?

A policy proposal was put to government to fund an action research program promoting collaboration right across the rural research sector. response pending.

Case study 3.
A team which had used the pattern to run a workshop on local response to climate change used the pattern process again for their de-briefing:
Focus question: *How best for a team to apply the collective social learning pattern in a social change workshop?*

What should be?

Team members answered "Clarity of purpose and shared interest in the outcome; ensure participants are clear about what they are there for and have faith in the process. Team members need to establish mutual respect, honesty in personal aims for the workshop and clear lines of responsibility.

What is?

Each knowledge culture's skills, experiences and goals need to be translated from conflicts of interest to trust and cooperation. Essentials are rules of dialogue, a peaceful ambience, and careful mutual listening. Accept that participants are likely to be competitive, individualised and alienated.

What could be?

A climate of creative imagination, and hopefulness; buzz of exciting new ideas; people profoundly catalysed to think; the process used as a replacement for action; individuals angry at having to share their mental space; time needed for reflection; people confident to express a range of very creative, very different, alternative, 'way out" ideas; making unusual links or connections.

What can be?

We can be a fantastic team, each working from our own skills base and at the same time in a collective team process; together we can bring change. We need to share our collective techniques/tricks eg 'learning circles'; strategic futures planing; learn from what happened but do it better; follow-up with a second series of vision workshops; do something differently with music and the arts.

Applications of the social learning spiral follow the same route for each of the widely varied wicked problems. Resolution requires collective social change, although the precise problem and outcome goal is quite different in each case. In each case the collective thinking process brought innovative ideas and integrated programs to put them into practice.

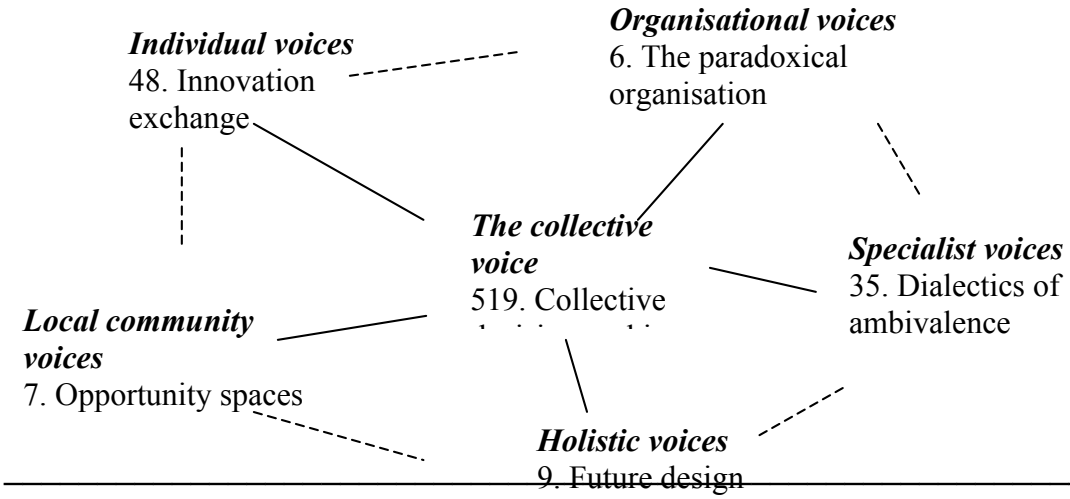
In Figure 2. the social learning spiral provides a framework for collective thinking and action. The knowledge cultures are now nested in a (a system of equivalent wholes); no longer a hierarchy. Each builds on each other. The various symbols reflect the contribution of each knowledge. Community knowledge is the almost invisible matrix which underpins each community's construction of reality. Communities are widely diverse, but link together into governmental regions and nations – hence the wavy line. Specialist knowledge in its turn draws on community experience by collecting data for the different specialist frameworks (the ring of boxes). Administrations and governments use community experience and the collected knowledge of the specialist disciplines in setting direction (the circle with arrows). The core or holistic centre of knowledge of the issues is a shared understanding of the whole.

While it doesn't matter where one enters the nested set, or even in what order they are drawn into a collective learning framework, it does matter that all the knowledges are equally respected and involved.

6. Related links

The website http://www.publicsphereproject.org/patterns/ contains several hundred patterns, making up a pattern language for *Liberating Voices: pattern languages for civic communication* (Schuler 2008). The present pattern is No. 519 in that collection and patterns 48, 7, 35, 6, and 9 give greater depth to the understanding of the individual, local community, specialised, organisational and holistic knowledge. Schummer's *Supporting social action in NGO'* assists social change practitioners from each of the knowledge cultures to mobilise existing skills and learn from each other through the rich resources of Web 2.

Figure 4. Collective decision-making cluster



7. Changed context: collecti social learning brings transformational change

Figure 3 places the collective knowledge within the collective social learning spiral The pattern was applied in the examples described above (regional agricultural development, an integrated research policy, and a social change workshop) and the context changed in each case.

In the case of sustainable agricultural development, the process brought together the nine industries of the region in a set of collaborative actions, possibly for the first time.The project led to acceptance of collective behavioural change framework. In the policy development project, the collective learning process revealed to the participants that they were working competitively even though they shared a common

goal. This lead to the development of a collaboration policy for the set of research and development corporations involved.

The information technology world has been changing too. The relational foundation for Google, the interactive capacity of Facebook and the validated knowledge in Wikipedia have changed our behaviour with respect to information searches, our friendship circles, and access to the construction of knowledge, respectively. Foucault has ably summed up the situation with: until the 20th century access to knowledge meant power; with the advent of information technology, access to the construction of knowledge meant power. The collective learning pattern gives the user access to that power.

> But does the collective social change pattern meet the conditions for solving wicked problems?
> 1. **No final solution**: The recognition that learning is cumulative and open-ended is represented by the spiral of collective social learning
> 2. **Every problem is unique**: The collective social learning approach is put into operation only with a given set of people at a specific time and place.
> 3. **Existing solutions impede essential changes**: Existing assumptions ideals, facts, ideas and actions are called into question at each stage of the learning cycle. Each sage of the cycle involves double-loop learning.
> 4. **Confusion between facts and values:** the distinction between fact and value is made explicit in following the learning spiral.
> **5: Solutions bring new problems:** The learning spiral assumes that each turn of the spiral will start afresh, facing the problems generated by and unsolved in the previous spiral.

In the debriefing of a team applying the collective social learning spiral, each team member came closer to appreciating the goals, factual basis, visioning processes and practical skills of the others. This led to a richer, more effective use of the pattern at each re-iteration.

Reviewing these events, it is becoming apparent that the collective learning pattern provides a general guideline for collaboratively redefining fragmented issues of long-standing. The pattern should therefore be of use to any one with responsibility for resolving a wicked problem, defined here as a complex problem arising from the actions of the society which produced it. This includes public health, environmental management, community development, organisational managers and change management practitioners.

Acknowledgements: I wish to acknowledge the inspired critiques of my shepherd, Linda Rising and the helpful comments of my writing group at EuroPLoP08

Bibliography:

Alexander, C. 2002b. *The nature of order. Book two: the process of creating life.* Berkeley, The Centre for Environmental Structure.

Alexander, C., Ishikawa, S., Silverstein, M. Jacobson, M., Fiksdahl-King, I. and Angel, S. 1977 *A pattern language*. New York: Oxford University Press.

Brown, VA 2007 *Leonardo's vision: a guide to collective thinking and action.* Rottterdam, SENSE Publishers.

Brown VA 2006 Towards the next Renaissance? Making collective decisions combining organisational, community and expert knowledge. *The International Journal of Knowledge, Culture and Change Management,* Vol. 6, Issue 3 pp 43-55.

Brown, V.A. 2001 Planners and the planet: reshaping the people/planet relationship: do planners have a role? *Australian Planner* 38 3 67- 73.

Google Search Engine http://www.google.com.au/ [30.2.07]

Haar, Jonathan *A Civil Action*

Keen, M., Brown, VA., and R. Dyball eds, 2005 *Social learning in environmental management. Towards a sustainable future*. Earthscan, London

Kolb, D.A., Lublin, S. and Spoth, J. 1986 Strategic management development: using experiential learning theory to asses and develop managerial competencies" *Journal of Management Development* (5) 3: 13-24.

Kolb, D. A., Rubin, I. M. & McIntyre, J.M. 1974 *Organizational psychology: an experiential approach*, Englewood Cliffs, Prentice Hall.

Schuler, D. 2008 *Liberating Voices: a pattern language for civic communication*. MIT Press, Massachusetts..

United Nations Conference on Environment and Development (UNCED). *Local Agenda 21: Chapter 28 of Agenda 21.* New York: Commission for Sustainable Development, 1993.

United Nations Environment Program (UNEP). 1972 World Conference on Environment (Stockholm Conference),

WCED (World Commission on Environment and Development) *Our Common Future : Report of the World Commission on Environment and Development.* Oxford: Oxford University Press. 1986

WHO (World Health Organization). The Ottawa Charter for Health Promotion. *Int. J. Health Promotion* 1986; 1(4):i-v.

WSSD (World Summit on Sustainable Development) 2002, *Concluding recommendations on partnerships* United Nations, Johannesburg

Modifiers: Increasing Richness and Nuance of Design Pattern Languages

Gwendolyn Kolfschoten[1] & Robert O. Briggs[2]

[1]Department of Systems Engineering
Faculty of Technology Policy and Management
Delft University of Technology
G.L.Kolfschoten@tudelft.nl

[2] Department of Business administration
Institute for Collaboration Science
University of Nebraska at Omaha
rbriggs@mail.unomaha.edu

May 7th 2009

Abstract

One of the challenges when establishing and maintaining a pattern language is to balance richness with simplicity. On the one hand, designers need a variety of useful design patterns to increase the speed of their design efforts and to reduce design risk. On the other hand, the greater the variety of design patterns in a language, the higher will be the cognitive load to remember and select among them. A solution to this is the modifier. The modifier concept emerged in a relatively new design pattern language called, ThinkLets. When analyzing the thinkLet pattern language we found that many of the patterns we knew were variations on other patterns. However, we also found patterns in these variation; we found variations that could be applied to different patterns, with similar effects. We document these variations as modifiers. In this paper we will explain the modifier concept, and show how they are not design patterns by themselves, they offer no solution by itself, and yet they produce predictable variations to a set of design patterns.

Introduction

Design patterns were first described by Alexander [Alexander 1979] as re-usable solutions to address frequently occurring problems. In Alexander's words: "a [design] pattern describes a problem which occurs over and over again and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice"[Alexander 1979 p x]. After the gang of four created a pattern language for software engineering [Gamma, Helm et al. 1995], the concept made its way in a variety of domains including collaboration support. For example, Lukosch and Schümmer [2006] propose a pattern language for the development of collaborative software. Design patterns are successfully used in related fields such as communication software [Rising 2001], e-learning [Niegemann and Domagk 2005] and for knowledge management [May and Taylor 2003]. Design patterns have various functions; they offer designers the building blocks for

their design effort, the can be used to capture best practices, they support teaching and training and they become a shared language among the users of the design patterns.

One of the challenges when establishing and maintaining a pattern language is to balance richness with simplicity. On the one hand, designers need a variety of useful design patterns to increase the speed of their design efforts and to reduce design risk. A pattern language with a greater number of design patterns offers a larger variety for inspiration and choice. With more relevant options, there are more circumstances where it is possible to establish a good fit between the design problem and design-pattern-based solution. On the other hand, the greater the variety of design patterns in a language, the higher will be the cognitive load to remember and select among them. A community of practice for a pattern language must therefore strive for parsimony. The community must attempt to capture useful and important design patterns, while keeping the design patterns consistent, coherent, interlinked and perhaps most importantly, non-overlapping. The need for parsimony conflicts with the need for greater variety and utility.

Design pattern modifiers provide a useful device for maintaining the parsimony of a pattern language while adding richness and nuance to its range of possibilities. A *modifier* is a not a complete pattern, but rather is a named, documented variation that can be applied to a collection of patterns. Modifiers create a predictable, useful changes in the solution derived from any pattern to which the modifier is applied.

The modifier concept emerged in a relatively new design pattern language called, ThinkLets [Kolfschoten, Briggs et al. 2006; Vreede, Briggs et al. 2006], and the concept is, perhaps, most easily demonstrated using that pattern language as an example. *ThinkLets* is a pattern language for designing collaborative work practices. A ThinkLet is a named, scripted collaborative activity that moves a group toward its goals in predictable, repeatable ways. As with other pattern languages, ThinkLets are used as design patterns, as design documentation, as a language for discussing complex and subtle design choices, and as training devices for transferring designs to practitioners in organizations.

In this paper, we present a formal articulation of the modifier concept. We first explain the thinkLets concept in more detail. We then explain the origins and nature of the modifier concept, and argue its utility as an extension to a pattern language. Next we offer examples of modifiers in the context of the thinkLets pattern language. We illustrate the effect a modifier can have on the execution of thinkLets. Finally, we discuss the need for and value of the modifier concept in pattern languages in general.

2. The ThinkLets Pattern Language

ThinkLets were originally derived to document the techniques and best practices of expert facilitators. Facilitators are group-process professionals who design collaborative work practices and conduct the processes on behalf of groups. The ThinkLets pattern language became more-rigorously codified and refined with the advent of the newly emerging field of Collaboration Engineering. *Collaboration Engineering* is specialty within the field of Facilitation. It is an approach to designing collaborative work practices for high-value recurring tasks and deploying the designs to practitioners to execute for themselves without the ongoing intervention of facilitators [Briggs, Vreede et al. 2003]. ThinkLets

therefore serve three different user groups; practitioners, who use thinkLets as scripts to support group work, Facilitators, who use thinkLets to exchange best practices in facilitation and to transfer them to novices, and Collaboration Engineers; who use thinkLets to rigorously design collaborative work practices and supporting technology in order to transfer these to practitioners. For design and knowledge sharing different aspects of a thinkLet are important. For knowledge sharing it is critical to have an extensive description of what will happen as an effect of the thinkLet, and a script on how to create this effect. For design it is (in addition) important to understand the context and situations in which the thinkLet can be applied and how the thinkLet can be combined with other thinkLets. For both situations basic design pattern properties such as a catchy name a picture, and an overview of what it does is important. For more information about the thinkLet set and concept we refer to [Kolfschoten, Briggs et al. 2006; Vreede, Briggs et al. 2006; Kolfschoten and Houten 2007].

3. The Modifier Pattern

Context: When you are authoring a pattern language with a community and this pattern language seems to explode in size. New patterns are continuously found but many patterns are similar. Some of the similar patterns turn out to be just instantiations of other design patterns. Others, however are clearly not instantiations of other patterns but rather they are deliberate variations made by experts to create a usefully different solution.

> Early in the life of the ThinkLets pattern language, the utility of the concept gave rise to an explosion of design patterns. Designers quickly hit information overload, so researchers began work to see whether they could distill the burgeoning collection down to an essential set [Kolfschoten, Appelman et al. 2004]. This research revealed that a number of the early thinkLets were useful variations on more-fundamental patterns. Some of these variations were actually just instantiations of other thinkLets.

Problem: Your pattern language is growing in size and complexity and new patters show overlap with existing patterns. You need more consistency and parsimony in your pattern language and you want to clearly distinguish patterns that you can combine and patterns that are variations of other patterns.

> An example of a thinkLet that turned out to be only an instantiation of another thinkLet was a thinkLet for SWOT analysis, where participants brainstorm ideas in four categories; Strengths, Weaknesses, Opportunities and Threats. Although the use of these specific categories has particular advantages and benefits, it was, in essence, an instantiation of the LeafHopper thinkLet described in the appendix. Detailed analysis of the newly developing ThinkLets pattern language showed, however, that after the instantiation duplications had been removed from the pattern collection, there remained a number of cases where the same variation had been applied to a number of different thinkLets, with the same predictable effect on each thinkLet to which it was applied. These variations involved removing or adding certain behavior rules to an existing thinkLet to create a predictable variation in its effect. These rule-changes occurred in a similar way, across different thinkLets. Researchers captured and named these variations, giving rise to the modifier concept.

Solution: Modifiers are reusable variations that can be applied to a number of design patterns in order to create a predictable change or variation to the solutions specified by these patterns. Using modifiers we can add nuance to a set of basic design patterns without suffering a combinatorial explosion of the pattern language. When modifiers are distilled from a set of design patterns, the pattern language can become richer, as more combinations can be made from fewer elements. At the same time the pattern language becomes more concise as the number of concepts in the patter language becomes smaller.

We define modifiers as named changes-of-rules that can be applied to one or more thinkLets to create predictable variations to the pattern of collaboration a thinkLet invokes, and predictable variations in the structure and quality of the outcomes produced by pattern. Modifier documentation specifies the changes-of-rules that comprise the modifier, the thinkLets to which those changes can be applied, and insights about the effect of the changes-of-rules will have on patterns of collaboration and quality of outcomes. To summarize, modifiers have one or more of the following characteristics:

- They can add new rules or delete existing rules from a thinkLet.
- They can alter a rule in the thinkLet.
- They crate a variation on the emerging pattern of collaboration.
- They alter the structure and quality of group outcomes in predictable ways.

Example, simplifying and enriching a pattern language with modifiers:

We compared the underlying rules for 7 idea-generation (brainstorming) thinkLets [Kolfschoten and Santanen 2007]. The analysis revealed that, although there were superficial differences. Some of those thinkLets were, at the level of their rules, virtually identical. Thus, the 7 of thinkLets could be collapsed to an essential set of four. In that same set of seven thinkLets, however, we found 12 variations that could be abstracted, and then deliberately added to or removed from some set of thinkLets to create a specific variation in their effects. These differences we captured as twelve named modifiers. For example a modifier used for idea-generation, or brainstorming activities:

- **OneUp** – each contribution must be arguably better along some specified dimension of quality than the ideas that have already been contributed.

The OneUp modifier had three predictable effects on any idea-generation thinkLet to which it was applied: a) participants generated a greater number of high-quality ideas and a lower number of low-quality ideas; b) participants engaged in less discussion of the ideas of others; and c) participants were less sure at the end of the activity that others understood the ideas they had contributed.

Modifiers do not alter the general pattern of collaboration that a thinkLet invokes. Rather, they produce nuanced variations on those patterns. The OneUp modifier, for instance, places an additional constraint on basic rules of any idea generation activity.

The twelve modifiers we derived could be applied in various combinations to create variations on one or more of the four basic idea generation thinkLets [Kolfschoten, Appelman et al. 2004; Kolfschoten and Santanen 2007]. The four thinkLets and twelve modifiers that emerged could be combined in a total of 43 combinations. Without the modifier concept, the pattern language would have therefore required 43 thinkLets to capture those useful variations. With the modifier concept, the same design power can be obtained with only 16 concepts – the four thinkLets and twelve modifiers. Thus, by distilling out the thinkLets and modifiers out of the 7 thinkLets, we uncovered 43 new design possibilities, and yet maintained the parsimony of the pattern language at 16 components.

In one sense, a modifier is to a thinkLet as a virus is to a cell. A virus is not a living organism, because on its own it cannot respire, digest, or reproduce. A virus, however, invokes predictable changes on the way the cell performs. In like manner, modifiers are also not complete design patterns, because on their own, they cannot be used to invoke predictable, repeatable patterns of collaboration. Rather, they can be applied to thinkLets to create predictable *changes* in the patterns of collaboration the thinkLet invokes. Modifiers have less information in their documentation because they are not complete thinkLets. They are therefore easier to master than a full thinkLet, which further reduces the cognitive load of mastering the pattern language.

Example- ThinkLets and ThinkLet Modifiers

ThinkLets are design patterns that can be used to create patterns in how people collaborate and the type of result they will jointly produce. As such they are prescriptive. To explain the modifier concept we will first introduce a set of different thinkLets with very different effects, next we will describe 3 modifiers and show how they are applicable for the different thinkLets. Table 1 lists six thinkLets that can be used to invoke a specific effect[1]. Note that these descriptions are not complete design pattern documentation, but rather a brief overview of the collaboration technique, sufficient for the reader to understand the nature of the pattern.

[1] The effects of thinkLets are also called patterns of collaboration. These are descriptive patterns and explain how the group moves from one state to another state, see Briggs, R.O.; Kolfschoten, G.L.; Vreede, G.J. de and Dean, D.L. (2006). Defining Key Concepts for Collaboration Engineering, *Americas Conference on Information Systems*, Acapulco, Mexico, AIS.

Table 1. An Example of a ThinkLet for Each Pattern of Collaboration.		
ThinkLet Example	**Brief Summary of ThinkLet**	**Effect**
LeafHopper	All participants view a set of pages, one for each of several discussion topics. Each participant hops among the topics to add ideas as inspired by interest and expertise.	Generate
GoldMiner	Participants view a page containing a collection of ideas, perhaps from an earlier brainstorming activity. They work in parallel, moving the ideas they deem most worthy of more attention from the original page to another page	Reduce
Illuminator	Participants review a page of contributions for clarity. When a participant judges a contribution to be vague or ambiguous, s/he requests clarification. Other group members offer explanations, and the group agrees to a shared definition. If necessary, the group revises the contribution to better convey its agreed meaning.	Clarify
PopcornSort	Participants work in parallel to move ideas from an unorganized list into to labeled categories, using a first-come-first-served protocol for deciding who gets to move each idea into a category.	Organize
StrawPoll	Moderator posts a page of unevaluated contributions. Participants are instructed to rate each item on a designated scale using designated criteria. Participants are told that they are not making a decision, just getting a sense of the group's opinions to help focus subsequent discussion.	Evaluate
Crowbar	After a vote, the moderator draws the group's attention to the items with the most disagreement. Group members discuss the reasons why someone might give an item a high rating, and why someone might give the item a low rating. The resulting conversation reveals unchallenged assumptions, unshared information, conflicts of goals, and other information useful to moving toward consensus.	Build consensus

We will now describe three modifiers. Modifiers are not complete design patterns, but rather named, codified changes that can be applied to one or more thinkLets to create predictable changes in the function of the thinkLet. Modifier documentation includes the name, purpose, and rule-change for the modifier, and a short explanation of effects the modifier will create. In the thinkLet documentation we capture with which modifiers the thinkLets can be combined.

Table 2. Modifier examples	
One Up	Participants are instructed that each new contribution must be better than existing contributions according some specified criteria. For example, "Please give me an idea that is more flexible than those we already have, Please suggest an idea that would be cheaper…" This encourages the contribution of ideas with specific desired qualities [Grünbacher, Halling et al. 2004].
Identification	Let participants choose to identify their actions; e.g. author, editor, voter, deleter. Some thinkLets by default let groups act anonymously, which, in many cases, has a positive effect on willingness to contribute. Other times, however, it is useful to allow identification, which enables participants to receive credit for or be held accountable for their actions, or to emphasize their stake or role. [Valacich, Jessup et al. 1992].
Chauffeured	Instead of working in parallel, participants jointly decide what action should be taken; e.g. joint organizing, joint clarification/rephrasing, joint evaluation. One participant serves as chauffeur for the group, actually taking the action in accordance with the wishes of the group. While parallel work can be more efficient than chauffeured work, in some cases it is more valuable to ensure shared understanding and to reach mutually acceptable agreements than to it is to be more efficient.

Table 3. Implications of modifiers on different patterns of collaboration, and the thinkLets within it

	One up	Identification	chauffeured
Generate	Generate contributions that excel on specified criteria	Authorship of contributions / Editorship for changed contributions	People suggest contributions, a recorder writes them down
Reduce	Select /summarize to converge on contributions that excel on specified criteria	Authorship of abstractions, summaries, Identity of those who select or reject a contribution	Participants discuss which ideas are worthy of more attention, a chauffeur documents the choices
Clarify	Clarify how a new contribution exceeds others with respect to specified criteria	Lobbying, explanation from specific author's perspective	Participants discuss shared meaning, a recorder documents their decisions
Organize		Identify of those who create, change, or delete relationships among contributions	Participants discuss relationships among contributions, a chauffeur documents the relationships
Evaluate		Display of polling results by pseudonym, by role, or by participant.	Participants discuss the value of concepts toward goal attainment. A chauffeur records their evaluations.
Consensus building		Identity of people willing or unwilling to commit to a proposal vs. anonymous indications of willingness to commit	

In the table above we show how each modifier can be applied to thinkLets for each effect of collaboration. For instance one-up can be used when generating ideas to stimulate excellence of contributions on a specific criterion, but can be used for organizing in the same manner to stimulate that contributions are related based on a specific criterion. Identification can be used in combination with various patterns to encourage or enable participants to take ownership, and chauffeuring can be used across patterns to create buy-in and ownership of choices.

Discussion and Conclusions

We argue in this paper the need for and the value of the modifier concept in pattern languages. We demonstrate that modifiers can make a pattern language at once both more powerful and more parsimonious. Modifiers make a pattern language more powerful by extending the variety and nuance of the solutions the language models. Modifiers make a pattern language more parsimonious by reducing a tendency toward combinatorial explosion of design patterns. Modifiers by themselves are

not complete design patterns. Rather, they are named revisions that can be applied to a set of design patterns to create predictable variations in the solutions based on the design patterns.

We have demonstrated the value of modifiers in the ThinkLets pattern language. We found that the idea of variation exists in software design patterns as the 'refine' concept, defined by [Noble 1998] "A specific pattern refines a more abstract pattern if the specific pattern's full description is a direct extension of the more general pattern. That is, the specific pattern must deal with a specialisation of the problem the general pattern addresses, must have a similar (but more specialised) solution structure, and must address the same forces as the more general pattern, but may also address additional forces." This author, however, describes 'refine' , in terms closer to object-oriented inheritance than to a variation that can be applied across many patterns. Nonetheless, when multiple refine relations exist with a single design pattern, there is an opportunity to simplify the pattern language through the use of modifiers.

A similar phenomenon can be found in the work of Hvatum et al [Hvatum, Simien et al. 2005]. Their pattern language includes both design patterns and advice for the management of distributed development teams. Advice patterns are not intended to actually structure the work of team members, but rather they describe conditions required for the patterns to work. This is similar to the modifier concept and enables the pattern authors to keep their pattern language simple and yet rich with advice.

Finally we found an example of Modifiers in the famous pattern language of Coplien and Harrison [Coplien and Harrison 2005] on organizational design patterns. In this pattern language a key cornerstone is the pattern "community of trust" it explains how trust is the basis for successful teams and a requirement for various other patterns to work. However, on it's own trust does not prescribe how organizations should be designed, the way other patterns in the language prescribe how roles and tasks and collaboration can be designed to successfully design the organization. Coplien and Harrison discuss why "community of trust" is a pattern, they explain it has structural impact on the organizational design and that there is a specific 'trick' to build trust described in "community of trust". The modifier concept will allow them to further position trust as a variation to other patterns. In this way they can emphasize it's critical nature, it's effect in combination other patterns, and the effect of the absence of trust in other patterns.

While the use of modifiers may not be obvious or necessary in every domain, modifiers may help to both enrich and simplify some pattern languages, while offering a wider variety of useful and deliberate design choices.

Further research is required to evaluate the added value of the use of modifiers from both an expert perspective (authors of pattern languages) and from a user perspective (communities that use design patterns). Initial discussions with both were positive; authors see the value of this concept for their languages and users can give examples of patterns that could be described as modifiers.

Acknowledgements

We thank our shepherd Kristian Elof Sørensen for his insightful reviews. Further we thank Stephan Lukosch for introducing us to and encouraging our participation in the Plop community.

References

Alexander, C. (1979). *The Timeless Way of Building*, New York, Oxford University Press.

Briggs, R.O.; Kolfschoten, G.L.; Vreede, G.J. de and Dean, D.L. (2006). Defining Key Concepts for Collaboration Engineering, *Americas Conference on Information Systems*, Acapulco, Mexico, AIS.

Briggs, R.O.; Vreede, G.J. de and Nunamaker, J.F. jr (2003). Collaboration Engineering with ThinkLets to Pursue Sustained Success with Group Support Systems, *Journal of Management Information Systems* **19,**(4): 31-63.

Coplien, J.O. and Harrison, N.B. (2005). *Organizational Patterns of Agile Software Development*, Upper Saddle River, NJ, Pearson Prentice Hall.

Gamma, E.; Helm, R.; Johnson, R. and Vlissides, J. (1995). *Elements of Reusable Object-Oriented Software*, Addison-Wesley Publishing Company.

Grünbacher, P.; Halling, M.; Biffl, S.; Kitapchi, H. and Boehm, B.W. (2004). Integrating Collaborative Processes and Quality Assurance Techniques: Experiences from Requirements Negotiation, *Journal of Management Information Systems* **20,**(4): 9-29.

Hvatum, L.B.; Simien, T.; Cretoiu, A. and Hliot, D. (2005). Patterns and Advice for Managing Distributed Product Development Teams, *Euro Plop*, Irsee, Germany, EuroPlop.

Kolfschoten, G.L.; Appelman, J.H.; Briggs, R.O. and Vreede, G.J. de (2004). Recurring Patterns of Facilitation Interventions in GSS Sessions, *Hawaii International Conference on System Sciences*, Los Alamitos, IEEE Computer Society Press.

Kolfschoten, G.L.; Briggs, R.O.; Vreede, G.J., de; Jacobs, P.H.M. and Appelman, J.H. (2006). Conceptual Foundation of the ThinkLet Concept for Collaboration Engineering, *International Journal of Human Computer Science* **64,**(7): 611-621.

Kolfschoten, G.L. and Houten, S.P.A. van (2007). Predictable Patterns in Group Settings through the use of Rule Based Facilitation Interventions, *Group Decision and Negotiation conference*, Mt Tremblant, Concordia University.

Kolfschoten, G.L. and Santanen, E.L. (2007). Reconceptualizing Generate ThinkLets: the Role of the Modifier, *Hawaii International Conference on System Science*, Waikoloa, IEEE Computer Society Press.

Lukosch, S. and Schümmer, T. (2006). Groupware Development Support with Technology Patterns, *International Journal of Human Computer Systems* **64**.

May, D. and Taylor, P. (2003). Knowledge Management with Patterns: Developing techniques to improve the process of converting information to knowledge, *Communications of the ACM* **44,**(7): 94-99.

Niegemann, H.M. and Domagk, S. (2005). ELEN Project Evaluation Report, from http://www2tisip.no/E-LEN.

Noble, J. (1998). Classifying Relationships between Object-Oriented Design Patterns, *Australian Software Engineering Conference* IEEE Computer Society Press.

Rising, L. (2001). *Design Patterns in Communication Software*, Cambridge, Cambridge University Press.

Valacich, J.S.; Jessup, L.M.; Dennis, A.R. and Nunamaker, J.F. jr. (1992). A Conceptual Framework of Anonymity in Group Support Systems, *Group Decision and Negotiation* **1**: 219-241.

Vreede, G.J. de; Briggs, R.O. and Kolfschoten, G.L. (2006). ThinkLets: A Pattern Language for Facilitated and Practitioner-Guided Collaboration Processes, *International Journal of Computer Applications in Technology* **25,**(2/3): 140-154.

The Role of Roles in Computer-mediated Interaction

Stephan Lukosch
Delft University of Technology
Faculty of Technology, Policy, and Management
PO box 5015, 2600 GA Delft, The Netherlands
s.g.lukosch@tudelft.nl

Till Schümmer
FernUniversität in Hagen
Department for Mathematics and Computer Science
Universitätsstr. 1, 58084 Hagen, Germany
till.schuemmer@fernuni-hagen.de

Abstract

Roles coin any social interaction. In this paper, we present basic practices for designing roles in collaboration settings. These patterns should help the designer of collaborative systems to reflect roles in the system design and thereby steer group interaction.

1 Introduction

The concept of a role is omnipresent in any interaction. This paper, e.g., has been written by two humans who took the role of the author. After one author created an initial draft of one section, the other author took the role of a devil's advocate. He questioned the theses of the first author and thereby helped him to clarify his point. When we submitted the paper, it was received by two people playing the role of a conference and a programme chair of EuroPLoP 2008. They checked the formal content of the paper and passed it on to a group of 10 people who were in the role of a programme committee. These people were expected to provide an assessment of the paper and judge whether or not it could be raised to a sufficient quality during a shepherding process. The programme committee

Proceedings of the 13th European Conference on Pattern Languages of Programs (EuroPLoP 2008), edited by Till Schümmer and Allan Kelly, ISSN 1613-0073 <issn-1613-0073.html>.

members passed their feedback back to the programme chair who released this paper for shepherding. At this point another group came into play: A pool of shepherds scanned this paper and decided whether or not they wanted to play the role of a shepherd. The shepherd's responsibility was to point out strong and weak points of the paper and help the authors to improve the weak points.

We could continue this story for one or two additional pages and thereby outline the interaction process that supported the evolution of this paper. From the example, we can already identify the core of the role concept: *A role combines prototypical behavior, rights, capabilities, and obligations.* Compared to this, tasks are expected activities. They are related to roles in workflows: From a rather abstract viewpoint which is sufficient for this paper, a workflow describes a sequence (or network) of tasks and relates them with roles. When enacting a workflow, the roles will be filled by concrete users who are then obliged to perform the task in a given time frame.

At the beginning of the process of our example, the author is expected to behave in a way that he writes the text of the paper, he has the right of expressing his thoughts. This however implies cognitive capabilities (e.g., the ability of formulating sentences or the capability of synthesizing new lines of thought) as well as technical capabilities like the access to a word processor or pen and paper. Finally, the author has the obligation of delivering the text in time with a sufficient quality.

The above example shows that a role owner has a different perception of the role as persons that are expecting specific activities from the role owner. From a philosophical perspective, Mead (1934) explored the two sides of the self, the 'me' as the social self and the 'I' as a response to the 'me'. In addition to Mead's theory, Cronk (2005) points out:

> There is a dialectical relationship between society and the individual; and this dialectic is enacted on the intra-psychic level in terms of the polarity of the 'me' and the 'I'. The 'me' is the internalization of roles which derive from such symbolic processes as linguistic interaction, playing, and gaming; whereas the 'I' is a 'creative response' to the symbolized structures of the 'me' (i.e., to the generalized other). (Cronk 2005)

This makes clear that there is always a dialogue between the role that contributes to the "me" and the "I" that constitutes the specific moment and the actions. Depending on the context of the self, there can be more or less need for creativity. If the human participates in a *strict workflow* or *production workflow* (Borghoff and Schlichter 2000) as we know it from *workflow management systems*, creativity is not desired in the participant's response. System design for such interaction should thus codify roles and restrict the capabilities of the user to actions that contribute to the expected behavior.

Looking at real collaboration scenarios, however, often shows a different style of interaction. Bardram (1997), e.g., investigated collaboration workflows in hospital settings and showed that plans are often created in-situ. Patients and doctors adapt their behavior so that it fits with the current situation. Environments for ill-structured tasks that are difficult to describe independently of current actions and that require agile decisions about future actions, often rely on social roles and

at the same time provide all users with capabilities that extend the capabilities of their specific role. This allows participants to diverge from their pre-defined roles as needed. In the extreme case, technology support for roles is not required and coordination of expectations is supported by awareness mechanisms (e.g., a REMOTE SELECTION$_{\rightarrow \text{P4CMI}}$ that can be used to communicate on which artifacts a user is currently working).

In this paper, we present patterns for an intermediate understanding of the role concept: Designs in which roles are explicitly modeled while still providing space for divergence from the role (at least in some patterns).

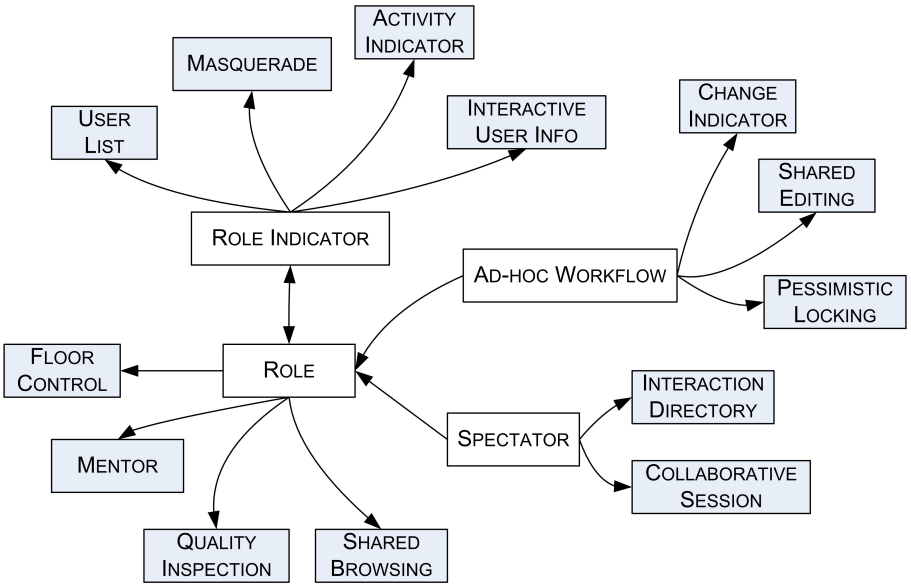


Figure 1: Pattern map

The patterns in this paper extend the 72 patterns for computer-mediated interaction that we present in (Schümmer and Lukosch 2007). Figure 1 shows the relations between the patterns in this paper and the patterns for computer-mediated interaction. The patterns for computer-mediated interaction follow the assumption that roles are part of the social agreement between the users of a collaborative system. In this paper, we frequently reference these patterns. Such a reference will be indicated by $_{\rightarrow \text{P4CMI}}$ next to a pattern name. Thumbnails of the referenced patterns can be found at the end of this paper. The four patterns in this paper are:

- ROLE: Explicitly model the responsibilities and capabilities of a role.
- ROLE INDICATOR: Decorate the VIRTUAL ME of a user with a symbol that represents the user's current role.
- AD-HOC WORKFLOW: Communicate dependencies between tasks and roles.
- SPECTATOR: Allow users to observe other users' actions.

While the first two patterns ease the understanding of the individual's capabilities and responsibilities, the third pattern supports the group's reflection on the interaction process. The last pattern finally helps to gain an outsider's view of the group process and is an example of a more concrete ROLE.

2 The Patterns

2.1 Role

Photo: Tyler,
http://www.flickr.com/photos/trp0/406897468/

Intent Model the expected interaction in the collaborative application.

Context You designed a system for computer-mediated interaction that shall support a specific group process.

Problem **Users have problems to structure their interaction in the group. Especially, some users act in a way that is not anticipated by other users. This can hinder the interaction to reach the intended goal.**

Scenario John and Paul interact in a software project. They would like to do an XP session using an application sharing tool, but it is pure chaos. Both start typing text, both want to program as the inspiration enters their minds. They are in trance and totally ignore the presence of each other, as there is no group process which structures the interaction. As a result, no real collaboration takes place and numerous conflicts evolve.

Symptoms *You should consider to apply the pattern when ...*

- collaboration needs supervision and guidance.
- there are administrative tasks to do that require deeper understanding of the consequences.
- some users are unaware of the group process and do not behave according to the other users' expectations.
- users see features that they should not or cannot use.
- users repeatedly assign a comparable combination of access rights to different users.

Solution **Therefore: Define roles that describe what the owner of the role is supposed to do. Also specify in a role which tools may be used in order to reach the role's intended**

goal. Link the roles to users when they engage in the group process.

Collaborations

Represent ROLES in your collaborative system. Each ROLE has a description and is associated to activities that can be performed by the person having this role.

Users can perform all activities that are available in their roles and necessary to complete the assigned task. Before an activity triggered by the user is executed, the system checks if the user owns a role that is associated to the activity. If not, no action will be performed.

There are two special roles: the omnipotent role allows a user to perform any action (an example is the role of an administrator) and the empty role is not related to any actions (this is equal to no role).

Roles can contain other roles. By that, the set of possible roles is combined. Users can play more than one role at the same time which means that they can execute any action allowed in any of their roles.

Roles can be assigned to and withdrawn for a user. In order to influence the user's actions, the user has to see what role he plays. This can be done by means of a ROLE INDICATOR$_{\rightarrow 2.2}$ or by sending a message to the users whenever they should switch their role.

It is most common to assign a default user role to an account at the moment the account is created. A system administrator can assign roles with advanced rights to user accounts afterwards (QUICK REGISTRATION$_{\rightarrow \text{P4CMI}}$). The system administrator role is assigned to a user account at the moment of system installation.

Rationale

The explicit notion of a role helps the users to understand their current situation in the group process. Since the role carries a description that explains what is expected from the person playing the role, it can help the users to fit their actions with the role.

The connection between role and action explicitly defines how a user can reach the role's goal. Assigning a user to the role ensures that all actions connected to the role can be executed by the user performing the role.

Assigning the allowed activities for a role instead of specific users reduces the amount of time spent on tool administration.

Check

When applying this pattern, you should answer these questions:

- What is the default or minimal role a user must have to act in the system?
- What role hierarchy should be implemented?
- Who is the authority to manage the association of roles to

users? Can users pass roles on?

- What are the events that give rise to a modification of the role?

Danger Spots

There are many reasons why users may fail to fill their role. They may be absent because of illness or they may lack competencies required to act in this role. For such cases, the system has to provide means for reassigning the role to another user.

Users may also abuse the power that is given by the role. Again, the system needs to provide mechanisms to revoke the role from such users.

Especially in creative processes, roles cannot be pre-defined. The definition of roles may be impossible at all, though ROLES might emerge from interaction implicitly. The AD-HOC WORKFLOW$_{\neg 2.3}$ pattern discusses the role of roles in such contexts.

Known Uses

Scripted Exercises in CURE (Haake 2007): In the context of computer-supported collaborative learning, scripts have been used to guide students through the exercises. The students were asked to first brainstorm concepts learned in the course, cluster the material, and finally write an essay on the topic. The essay writing process was supported by two roles: The author was asked to create a draft of an essay. Then, the other group members took the role of a reviewer and annotated the initial essay. After receiving the reviews, the author could modify the text or pass his role on to another group member and become a reviewer instead.

Blackboard: In Blackboard (Blackboard Inc. 2009), each user role has a specific set of permission levels:

- Course Builder - has access to all features except Assessments and Course Tools.
- Grader - has access to the grade book and is able to create and modify assessments.
- Instructor - has access to all course functions. This includes adding and modifying content, controlling user and group functions, creating assessments and entering grades, and controlling discussion boards and virtual classroom functions.
- Student - has access to all course content but cannot modify content. This is the only role able to take assessments and have grades recorded in the grade book.
- Teacher Assistant - shares the same level of access as the instructor. Although the rights are the same, the name of the role conjures different expectations regarding the teacher's behavior.

The admin assigns a new role to the selected user account.

Many other CSCL systems work the same way, e.g. Moodle (Moodle 2009), ILIAS (ILIAS 2009), and Synergeia (Stahl 2002).

World of Warcraft (`http://www.wow-europe.com/`) is a distributed multi-user game in which teams can play together against other teams. To initiate a team, one user invites other users to the team. The inviting user will have the role of a team leader and has special rights such as inviting new players to the team or deciding on how the goods are shared among the team members.

SourceForge.net is an open source software development web site in which project members can have different ROLES, e.g. administrator, developer, translator, etc.

Related Patterns QUALITY INSPECTION$_{\rightarrow \text{P4CMI}}$, MENTOR$_{\rightarrow \text{P4CMI}}$, SHARED BROWSING$_{\rightarrow \text{P4CMI}}$ are examples of patterns that rely on roles. These patterns focus on the group process and show how specific user roles can be supported.

FLOOR CONTROL$_{\rightarrow \text{P4CMI}}$ describes how roles can be passed on to other users.

ROLE INDICATOR$_{\rightarrow 2.2}$ shows which role a user currently has.

ROLE-BASED ACCESS CONTROL (Schumacher et al. 2005) discusses the issue of access rights in relation to roles.

2.2 Role Indicator

Photo: Ron Bird, FreeDigitalPhotos.net

Intent Let each member of a group know which Roles the other members have.

Context You are developing a computer-mediated environment, where users can have different Roles with different rights.

Problem **Users are not aware of capabilities as well as responsibilities of other users.**

Scenario Consider a globally distributed development project in which a large team co-constructs a game engine together with some test users. Molo, one of the African test users, has problems installing the new version of the game engine. Unfortunately, the water supply simulation game which he uses for testing does no longer work on top of the newest game engine. He would have liked to talk to the test officer in the project, but he does not know who this person currently is.

Symptoms *You should consider to apply the pattern when ...*

- Users want to perform a specific activity but are not allowed to do this.
- Users frequently ask someone else to perform specific activities.
- Users do not know who has enough rights to perform specific activities.
- Users treat other users as if they have a different role.

Solution **Therefore: Visualize the Role$_{\rightarrow 2.1}$ of the interacting users whenever a user is shown in the user interface.**

Collaborations Integrate an element in the User List$_{\rightarrow \text{P4CMI}}$ or the Interactive User Info$_{\rightarrow \text{P4CMI}}$ so that the current role of a user is revealed. Make sure that the role representation is unique and can be understood by all interacting users. When users can change their role,

update the role information whenever a user switches to another role.

Rationale As each user's role is visualized, users can easily lookup their own role in the interaction process and also identify the role of their peer users. This allows users to interact with each other according their roles.

Check *When applying this pattern, you should answer these questions:*

- What are the different ROLES?
- Where are you going to visualize the ROLES?
- How are you going to visualize the different ROLES? Are you going to use different icons for each ROLE or will you use textual labels?

Danger Spots Users might not want that their role is revealed. In such cases, you should allow users to turn their role indicator off.

Users might have different roles at the same time. This makes it difficult to decide which role is shown to the other users.

Known Uses **World of Warcraft** (`http://www.wow-europe.com/`) requires that each team has one leader. The leader has special rights, e.g. a leader can decide how rewards are distributed in the team or a leader may invite new team members. Leaders can be identified in the USER LIST$_{\rightarrow P4CMI}$ as their representation is associated with a small crown (cf. Figure 2).

Figure 2: ROLE INDICATORS in World of Warcraft

Vitero `http://www.vitero.de` is a conferencing system which supports up to two moderators. The moderators can pass

a microphone icon to user which want to talk. The microphone is shown to all other users as well so that they stay aware of who has currently the speaker role.

XPairtise (Lukosch and Schümmer 2007) is a tool for distributed pair programming. For supporting and teaching distributed pair programming, XPairtise distinguishes three different roles, i.e. navigator, driver, and SPECTATOR$_{\rightarrow 2.4}$. Figure 3 shows how the different roles are indicated in the USER LIST$_{\rightarrow \text{P4CMI}}$.

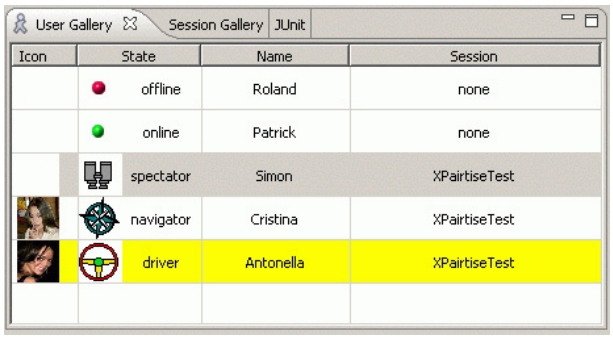


Figure 3: ROLE INDICATOR in Xpairtise

Related Patterns ROLE$_{\rightarrow 2.1}$**:** ROLE INDICATOR describes where and when to visualize different ROLES.

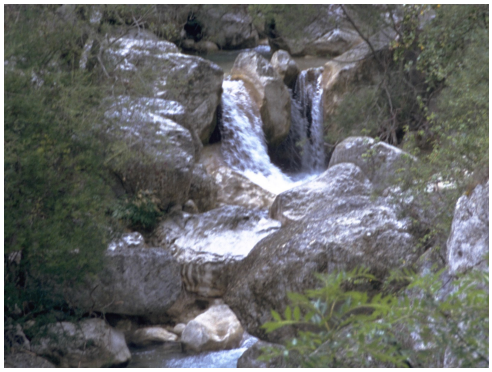
USER LIST$_{\rightarrow \text{P4CMI}}$ allows to visualize different ROLES by simply extending the user representation in the list.

MASQUERADE$_{\rightarrow \text{P4CMI}}$ describes how users can control what kind of personal information they reveal to other users. This can include information about a user's ROLE.

ACTIVITY INDICATOR$_{\rightarrow \text{P4CMI}}$ shows for awareness purposes the activities of the collaborating users. By analyzing the activities of a user, it is also possible to identify a user's role in the collaboration process.

INTERACTIVE USER INFO$_{\rightarrow \text{P4CMI}}$ equips a user representation with a context menu that allows to start an interaction with the represented user. It can easily be used to visualize the user's current ROLE.

2.3 Ad-hoc Workflow



AKA — Interaction Script

Intent — Communicate and make explicit dependencies between tasks and roles.

Context — You are interacting in a creative, ill-structured group process.

Problem — **Plans are a good thing. Having well-defined roles and tasks creates safety in performing tasks. However, ill-structured group processes are highly non-deterministic which means that they cannot be pressed in pre-defined task schemas. Plans will fail in most of these projects.**

Scenario — Consider a typical XP project. Linea, the customer created a set of task cards and arranged them in a lovely sequence. She created a picture of the project and believes that the team will be able to create a good solution by simply implementing the tasks. But already after the first task was done, the developers feel that the plan does not fit the context and after her first tests with the resulting system, Linea also feels the need to change the plan.

Symptoms — *You should consider to apply the pattern when ...*

- strict workflows are too strict for dynamic/creative group processes.
- users are not aware of the intended group process and their future tasks.
- users expect others to do their work.
- users do not understand the dependencies between tasks.

Solution — **Therefore: Collaboratively create an explicit representation of tasks as a shared document and thereby achieve a shared understanding of ad-hoc plans that dynamically adapts to the current group process.**

Collaborations — In the simplest form, the group members can use a wiki to list the different tasks and responsibilities. For synchronous collaboration, the group members can also make use of a Shared Editor$_{\rightarrow \text{P4CMI}}$.

Users should be supported in creating shared tasks, creating relations between tasks, and assigning Roles to tasks. Visualize the representation in a way that users can understand and perform the workflow. In case of a textual representation the workflow follows the linear structure of the text, e.g. each task is a list item. In case of a visual representation, the workflow is represented as a directed graph. Use the visual workflow to document current steps and guide the group process but allow the group to adapt the process as soon as it is necessary.

Rationale

Wil van der Aalst et al. (1999) described the taxonomy of collaborative work as shown in Figure 4. They classify collaborative work along two dimensions: structuredness and the center of attention.

Work can be highly structured as it is the case in optimized production workflows or it can be inherently unstructured as it is often the case for collaborative problem solving activities. The support for the group interaction can focus on making information available to all group members (and providing the best comprehensible awareness on the group members' activities) or it can focus on supporting the interaction process and thus guide the group members through the required steps.

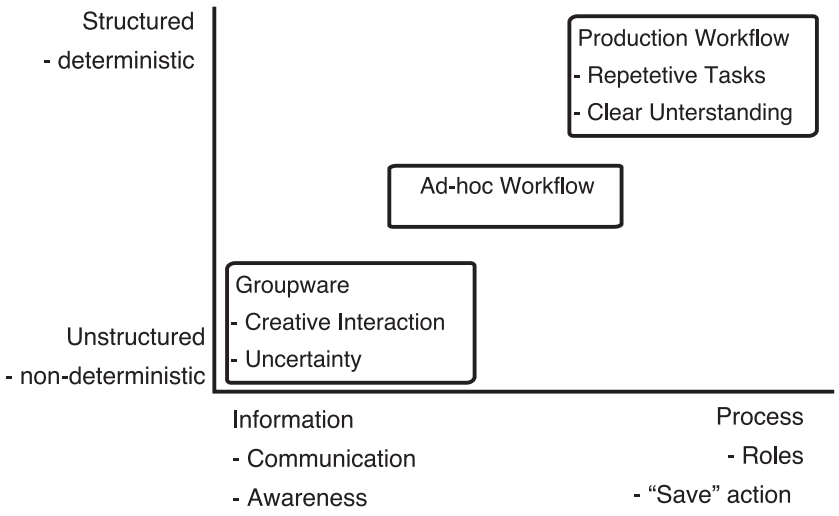


Figure 4: Situating collaborative work according to the structure and focus dimensions (inspired by van der Aalst et al. (1999)).

Groupware applications are typically situated in the lower left corner of the diagram: they support creative interaction and help a group to interact on shared information spaces. They focus on improving the communication between the group members and make them aware of each others' actions.

Workflow management systems on the other hand focus on guiding the users through the process. The process is pre-defined and the group members are only required to perform their steps in

the process. Communication is pre-structured and in most cases reduced to the communication acts required for executing the workflow.

An Ad-Hoc Workflow helps to structure the implicit processes of information-centered unstructured interaction. During the co-construction of the Ad-Hoc Workflow, the group members become aware of the required steps for the specific tasks and coordinate their efforts.

During the enactment phase of the workflow, the group members document their progress in the process and thereby increase the awareness of the group's activities. Since group members are allowed to deviate from the Ad-Hoc Workflow, they keep the flexibility of information-centered collaboration.

Check

When applying this pattern, you should answer these questions:

- When will you create the workflow in your group process? Can you distinguish coordination phases from collaboration phases?
- Will you create a graphical or a textual representation of the workflow?
- How do you visualize active tasks?
- Can you use the workflow representation to track your work?

Danger Spots

Workflows are often not just a a linear sequence. Especially for iterative processes, workflows may include iterations or parallel processing streams. The user has to be careful that no circular dependencies occur that may lead to deadlocks. In a deadlock situation, user A waits for user B to complete task 1 before A can start task 2. At the same time, user B waits for user a to complete task 2 before he can start task 1. In general, the groupware system should highlight potential deadlocks in the workflow. This can be done by checking the following deadlock conditions that are common knowledge in operating systems research and visualizing those tasks for which the conditions apply:

Mutual exclusion: the tasks assume that the performer of the task has exclusive access to a shared resource (see Pessimistic Locking$_{\rightarrow \text{P4CMI}}$).

Hold and wait: tasks require more than one shared resource. Once the performers have obtained a Pessimistic Lock$_{\rightarrow \text{P4CMI}}$, they keep the lock and request another lock to complete the task.

No preemption: there is no way to force a performer of a task to give back *his* resources.

Circular wait: task have a circular dependency as outlined in the previous paragraph.

Changes to the workflow can place another burden on the users. Once the ad-hoc workflow was changed, the users have to understand the new group process and adapt their behavior to act according to their role. Changes to the workflow should thus be highlighted, e.g., by placing a CHANGE INDICATOR$_{\rightarrow P4CMI}$ on the changed sections.

Known Uses

Chips / XChips (Rubart et al. 2001) visualized ad-hoc workflows as graph structures. The users could define task graphs and relate tasks with roles. When enacting the ad-hoc workflow, the users can assign group members to roles and thereby express responsibilities of individuals for specific tasks.

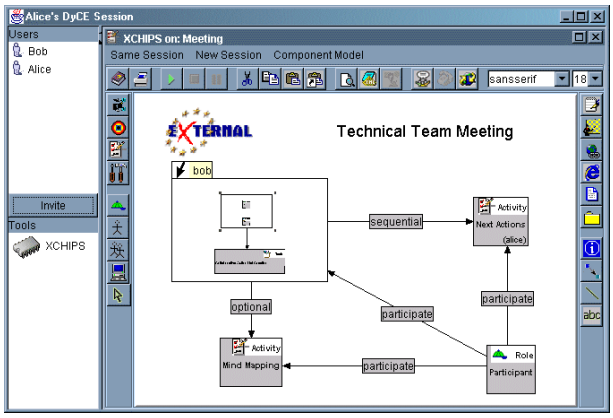


Figure 5: AD-HOC WORKFLOW in xChips

Figure 5 (Rubart and Haake 2003) shows how XChips was used to create an AD-HOC WORKFLOW for a company specific meeting (picture reprinted with authors' permissions).

DigiMod (`http://www.teambits.de/`) is a meeting facilitation support system that allows facilitators to structure the different phases of the meeting. Facilitators create tasks that describe what the participants should discuss and how the discussion should take place (e.g., as a structured brainstorming). During the meeting, DigiMod visualized the sequence of meeting steps and highlights the current task.

Related Patterns

ROLE$_{\rightarrow 2.1}$**:** Tasks are performed by ROLES. In the definition phase of the AD-HOC WORKFLOW, the group members name (or create) roles and associate them with tasks.

NO AGENDA, NO MEETING argues that all meetings should have an agenda. The agenda is comparable to a linear AD-HOC WORKFLOW.

SHARED EDITING$_{\rightarrow P4CMI}$ The workflow representation should be created using a shared editor. This allows all group members to participate in the creation of the workflow and contribute their views of an optimal problem solving path.

2.4 Spectator

Photo: Federico Stevanin, FreeDigitalPhotos.net

Intent Allow users to observe the activities of other interacting users.

Context Users are interacting in a computer-mediated environment to achieve a shared goal.

Problem **Users are interacting in a computer-mediated environment but are not familiar with the environment. These users perform activities which disturb the interaction and collaboration of other users.**

Scenario John and Paul have now managed to work in pair programming sessions. Now, their company is growing as it acquires more and more projects. The new employees are requested to work in pair programming sessions as well to ensure the high software quality, but they do not know how to do this. The manager requests John and Paul to teach the new employees, but their tool has only been designed to support one driver and one navigator in a pair programming session. Thus, John and Paul do not know how they can show the new employees how to interact in pair programming sessions.

Symptoms *You should consider to apply the pattern when ...*

- Unexperienced users are not accepted by experienced users.
- Unexperienced users disturb the collaboration of other users.
- Users want to communicate their experience with a computer-mediated environment but do not know how.

Solution **Therefore: Allow users to view and follow the interaction in an ongoing** Collaborative Session$_{\rightarrow \text{P4CMI}}$ **as** Spectator**. Ensure that these** Spectators **cannot influence the interaction.**

Collaborations Allow users to select an active Collaborative Session$_{\rightarrow \text{P4CMI}}$ from an Interaction Directory$_{\rightarrow \text{P4CMI}}$ and to choose whether

they want to join the session as a regular participant or as a Spectator. When joining as Spectator users can freely navigate in the shared artifacts which are used in the session. They can also view the activities of the other regular participants but they cannot influence the activities by modifying the shared artifacts as well.

Rationale

The Spectator Role does not allow that users influence the activities of other users. However, Spectators can view and understand the interaction of other users and thereby learn how to interact in the computer-mediated environment.

Check

When applying this pattern, you should answer these questions:

- Are you going to inform regular participants in a Collaborative Session about Spectators viewing their interaction?
- Is informing the regular participants enough or should they in some cases be asked for explicit permission?
- Are Spectators allowed to contact regular participants or other Spectators?
- Are you going to provide mechanisms for Spectators which make them aware of the other users' activities?

Danger Spots

Ensure that Spectators cannot access private artifacts of other users. You may even consider Role-based Access Control (Schumacher et al. 2005) to decide which artifacts Spectators or other roles may access.

Known Uses

Counter Strike Source (`http://store.steampowered.com/app/240/`) is a multi-user game in which teams compete with each other in Collaborative Sessions$_{\rightarrow \text{P4CMI}}$. Before participating in such a Collaborative Session, players have to decide which team they want to join. As additional opportunity, players can decide to join as a Spectator. Spectators can move around like regular participants but cannot influence the current game. For that purpose, players can switch their Role$_{\rightarrow 2.1}$ and become a regular participant.

Guild Wars is a MMORPG (Massively multiplayer online role-playing game) which apart role playing supports team competitions. The team competitions can be viewed by Spectators.

XPairtise (Lukosch and Schümmer 2007) is a tool for distributed pair programming. Apart from the Roles of a driver and navigator, XPairtise also supports Spectators which can follow an ongoing distributed pair programming session. When joining a pair programming session, users can choose between the different supported roles (cf. Figure 6).

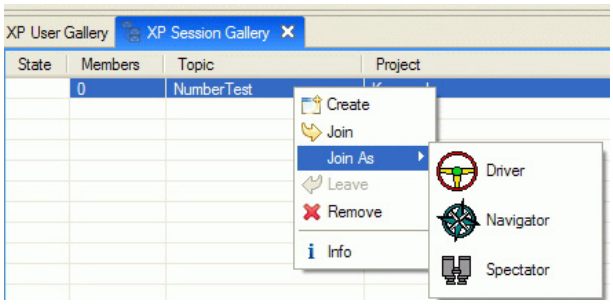


Figure 6: Role-dependent join in XPairtise

Related Patterns COLLABORATIVE SESSION$_{\rightarrow \text{P4CMI}}$: SPECTATORS can view the interaction in a COLLABORATIVE SESSION.

ROLE$_{\rightarrow 2.1}$: A SPECTATOR has a concrete ROLE.

ROLE-BASED ACCESS CONTROL (Schumacher et al. 2005) discusses the issue of access rights in relation to roles and can be used to define which artifacts a SPECTATOR may access.

3 Conclusions

The patterns of this papers discussed the role of roles in computer-mediated interaction settings. We presented a small selection of proven practices for modeling roles in such settings. However, we are aware of the fact that these patterns can only be a starting point towards a larger collection of practices that help groups to structure their group processes. It still needs to be investigated to what extent these practices can and should be written as patterns.

In some contexts, more domain-specific patterns have shown to be very helpful for practitioners of that specific domain. One example is the meeting patterns collection (Schuemmer and Tandler 2008) that names concrete roles in a group meeting, such as the role of the facilitator or the presenter. We foresee that more domain-specific pattern collections will emerge – and if there is no concrete collection for your specific domain, you may consider taking the role of an author and share your collaboration experience. May the patterns of this collection help you to structure your concrete patterns as well as concrete application that support your domain.

Acknowledgments

We would like to thank our shepherd Andreas Fießer for his excellent and challenging questions that helped to improve the patterns in this paper.

References

Bardram, J. E. (1997). Plans as situated action: an activity theory approach to workflow systems. In *ECSCW'97: Proceedings of the fifth conference on European Conference on Computer-Supported Cooperative Work*, Norwell, MA, USA, pp. 17–32. Kluwer Academic Publishers.

Blackboard Inc. (2009, February). Blackboard home. http://www.blackboard.com/.

Borghoff, U. M. and J. H. Schlichter (2000). *Computer-Supported Cooperative Work*. Springer-Verlag Berlin Heidelberg New York.

Cronk, G. (2005). George Herbert Mead – The Internet Encyclopedia of Philosophy. http://www.utm.edu/research/iep/m/mead.htm.

Haake, J. M. (2007). Computer-Supported Collaborative Scripts: Einsatz computergestützter Kooperationsskripte in der Fernlehre. In *DeLFI 2007, 5. e-Learning Fachtagung Informatik*, pp. 9–20.

ILIAS (2009, February). ILIAS open source LMS. http://www.ilias.de/.

Lukosch, S. and T. Schümmer (2007, September). Enabling distributed pair programming in Eclipse. In *10th European Conference on Computer-Supported Cooperative Work (ECSCW'07), Workshop 'The Challenges of Collaborative Work in Global Software Development'*.

Mead, G. H. (1934). *Mind, Self, and Society*. The Chicago University Press, Ltd., London.

Moodle (2009, February). Moodle.org: open-source community-based tools for learning. http://moodle.org/.

Rubart, J., J. M. Haake, D. A. Tietze, and W. Wang (2001). Organizing shared enterprise workspaces using component-based cooperative hypermedia. In *HYPERTEXT '01: Proceedings of the 12th ACM conference on Hypertext and Hypermedia*, New York, NY, USA, pp. 73–82. ACM.

Rubart, J., W. W. and J. M. Haake (2003). Supporting cooperative activities with shared hypermedia workspaces on the www. In *Alternate Track Proceedings of WWW 2003*. MTA SZTAKI.

Schuemmer, T. and P. Tandler (2008). Patterns for technology enhanced meetings. In *Proceedings of EuroPLOP'07*, Konstranz, Germany. UVK, Konstanz.

Schumacher, M., E. Fernandez-Buglioni, D. Hybertson, F. Buschmann, and P. Sommerlad (2005). *Security Patterns*. Chichester, UK: Wiley.

Schümmer, T. and S. Lukosch (2007). *Patterns for Computer-Mediated Interaction*. John Wiley & Sons, Ltd.

Stahl, G. (2002, September). Groupware goes to school. In J. M. Haake and J. A. Pino (Eds.), *Groupware: Design, Implementation, and Use, 8th International Workshop, CRIWG 2002*, LNCS 2440, La Serena, Chile, pp. 7–24. Springer-Verlag Berlin Heidelberg.

van der Aalst, W. M. P., T. Basten, H. M. W. Verbeek, P. A. C. Verkoulen, and M. Voorhoeve (1999). Adaptive workflow-on the interplay between flexibility and support. In *International Conference on Enterprise Information Systems*, pp. 353–360.

Appendix: Pattern Thumbnails

Activity Indicator

Problem: Users need time to perform a task but only the results are shared among them. In a collocated setting users are accustomed to perceive non-verbal signals such as movement or sounds when another user is active. If the users are distributed, these signals are missing. Users are therefore not aware of other users' activities, which can result in conflicting work or unnecessary delays.

Solution: Indicate other user's current activities in the user interface. To reduce interruptions, use a peripheral place or a visually unobtrusive indicator.

Change Indicator

Problem: While users works on independent local copies of artifacts, their check-out frequency for the artifacts may be low. As a result, they may work on old

copies, which leads to potentially conflicting parallel changes. The conflict is worse if two parallel modifications have contradictory intentions.

Solution: Indicate whenever an artifact has been changed by an actor other than the local user. Show this information whenever the artifact or a reference to the artifact is shown on the screen. The information should contain details about the type of change and provide access to the new version of the artifact.

Collaborative Session

Problem: Users need a shared context for synchronous collaboration. Computer-mediated environments are neither concrete nor visible, however. This makes it difficult to define a shared context and thereby plan synchronous collaboration.

Solution: Model the context for synchronous collaboration as a shared session object. Visualize the session state and support users in starting, joining, leaving, and terminating the session. When users join a session, automatically start the necessary collaboration tools.

Floor Control

Problem: Synchronous interaction can lead to parallel and conflicting actions that confuse the interacting users and makes interaction difficult.

Solution: Model the right to interact in the shared collaboration space by means of a token and only let the user holding the token modify or access the shared resources. Establish a fair group process for passing the token among interacting users.

Interaction Directory

Problem: Finding existing contexts to start interaction and memorizing older contexts to continue an interaction is difficult.

Solution: Provide a shared space that is available to all users in which users can store and retrieve interaction contexts.

Interactive User Info

Problem: Users are aware of other users in the collaboration space and can identify them, but they don't know how to start tighter interaction with a specific user.

Solution: Equip the user representation with a context menu that provides commands for finding out more information on a user and for starting tighter collaboration with the user.

Masquerade

Problem: Your application monitors the local user. The information gathered is used to provide awareness information to remote users. While this is suitable in some situations, users often do not act as confidently if they know they are monitored. Users may feel a need to avoid providing any information to others.

Solution: Let users control what information is revealed from their personal details in a specific interaction context. This means that users must be able to filter the information that is revealed from their personal information. Remember to consider reciprocity.

Mentor

Problem: Newcomers do not know how community members normally act in specific situations. They are not used to practices that are frequently applied in the community.

Solution: Pair newcomers with experienced group members who act as mentors. Initially let newcomers observe their mentors, and gradually shift control to the newcomer.

Pessimistic Locking

Problem: You want to ensure that changes performed by the user are definitely applied, even if more than one user wants to modify the same shared object at the same time.

Solution: Let a site request and receive a distributed lock before it can change the shared state. The lock can have different grain sizes. The grain size of a lock determines how much of a shared data object, or of all shared data objects, can be modified after getting one lock. After performing the change, let the site release the lock, so that other sites can request and receive it for changing the shared state.

Quality Inspection

Problem: Members participate in a community to enjoy high-quality contributions from fellow members. However, not every contribution has the same quality. Low-quality contributions can annoy community members and distract their attention from high-quality gems.

Solution: Select users as moderators and let them release only relevant contributions into the community's interaction space. Give moderators the right to remove any contribution and to expel users from the community.

Remote Selection

Problem: Users select artifacts to start an action on the artifact. Selecting an artifact is considered as taking the artifact under personal control. Whenever two users select the same artifacts, this leads to coordination problems.

Solution: Show remote users' selections to a local user. Make sure that other users who are interested in a specific artifact are aware of all distributed co-workers who have selected the object.

Shared Browsing

Problem: Users have problems finding relevant information in a collaboration space. They often get lost.

Solution: Browse through the information space together. Provide a means for communication, and collaborative browsers that show the same information at each client's site.

Shared Editing

Problem: Users are sharing data for collaboration. The need to edit the shared data simultaneously emerges, but the shared single-user application does not allow concurrent editing.

Solution: Provide a shared editor in which users can manipulate the shared artifacts together. Ensure that state changes are instantly reflected in all other users' editors, and provide mechanisms that make users aware of each other.

User List

Problem: Users do not know with whom they do or could interact. Consequently, they do not have the feeling of interacting in a group.

Solution: Provide awareness in context. Visualize who currently is accessing an artifact or participating in a Collaborative Session$_{\rightarrow \text{P4CMI}}$. Ensure that the information is always valid.

Virtual Me

Problem: In a large user community, account names look similar. But users need to communicate their identity in order to interact with other users.

Solution: Allow the users to play theater! Provide them with means to create a virtual identity that represents them while they act in the system. Show the virtual identity when the user is active.

Sharing Day

Lotte De Rore, Monique Snoeck, Guido Dedene
LIRIS Group, Faculty of Business and Economics, K.U.Leuven
Naamsestraat 69, 3000 Leuven, Belgium
{Lotte.DeRore, Monique.Snoeck}@econ.kuleuven.be

1 Introduction

In the IT department of a bank and insurance company, when Susan from the loan department is confronted with a problem, she will ask the members of her team for help or maybe the colleague sitting next to her. But it will not occur to her that Mike from the insurance department might have had this same problem before...

Knowledge is a very important asset for a company. Several techniques exist to share knowledge within a company. This paper introduces one of these techniques: SHARING DAY. The pattern language first describes why and when to organize a SHARING DAY and subsequently how to organize such an event.

2 Pattern Language

2.1 Overview of the Patterns

SHARING DAY	Create an explicit time to bring people physically together at the same location in order to encourage knowledge sharing on a broad basis.
PALLET OF ACTIVITIES	Match the form of the SHARING DAY to the kinds of knowledge and purposes you wish to share.
DRY RUN	Perform a DRY RUN of the sessions to see whether the stories in the different sessions are attuned on each other and attuned to the audience.
VISIBLE INVOLVEMENT	Make visible that the company attaches great importance to the SHARING DAY and stress the personal benefit of attending the SHARING DAY.
CENTRALIZED INFORMATION POINT	Centralize all communication about the SHARING DAY in one point to conserve the coherence.
KEEP IT FUN	KEEP IT FUN to help people stay involved and alert all the time during the SHARING DAY.
PROVIDE A BACK-UP	PROVIDE A BACK-UP to prevent the SHARING DAY from falling apart if some of the contributors drop out right before the planned day.

Proceedings of the 13th European Conference on Pattern Languages of Programs (EuroPLoP 2008), edited by Till Schümmer and Allan Kelly, ISSN 1613-0073 <issn-1613-0073.html>.

2.2 SHARING DAY

The hierarchical structure typical in large companies divides employees in several divisions. In each of these divisions, we find people with the same positions and roles, confronted with the same problems. Within a group, we have a diversity of knowledge and experience.

Knowledge sharing about the profession only occurs between colleagues in their own surroundings (within their own team or division); there is no spontaneous knowledge sharing on a broad base.

Knowledge within a company is a very important asset. In order to share knowledge about the profession, a discussion forum does not work. The most plausible reason for this is that people need to find free time to search or contribute to the discussion forum. The communication on a discussion forum happens in an asynchronous way which is not the most effective way of communication [1]. A personal and direct way of communication (like face to face) is the most powerful way of communication [1]. Also, in a culture with high expectations with respect to quality of solutions and answers, people wonder whether their knowledge is good enough to share e.g. on a discussion forum.

The project driven way of working in a company does not encourage knowledge sharing between teams or divisions. Every activity is budgeted and when in need for information, employees ask themselves whether it is justified to make time to communicate with other people. In addition they often don't know who they should talk with. The deadline of a project is seen as the most important factor and as a consequence, people work in a deliverable driven way rather than to strive for a uniform way of working across projects.

However, for a company in expansion, the kind of projects will change. When a company is still small, almost all projects are rather small and limited to the context of a single division. But when a company expands, projects grow and happen in an organization-wide context, involving several divisions in one project.

Local networking within the same division is no longer sufficient. Knowledge sharing and information exchange between divisions is necessary. There needs to be a way to make all the aspects within a job visible. For example, how to deal with a process, a tool or communication? What does it mean to fulfill a particular job?

Therefore,
Create an explicit time to bring people physically together to the same location by organizing a SHARING DAY.

The main purpose of the SHARING DAY is to give people a chance to share experiences. But even more important than sharing knowledge is to encourage networking, to know who knows what might be more useful than knowing something yourself. It is therefore important to withdraw the people from their normal work context. By having the event off site, one can avoid interruptions that disturb the flow of the event (LOCATION, LOCATION, LOCATION [2]).

The practical organization of such a big event as a SHARING DAY should not be underestimated. It consumes a lot of time to organize a well structured event. To ensure that the activities are fitting the variety of goals of a SHARING DAY, you should build the program

using a PALLET OF ACTIVITIES. A DRY RUN can help to see whether the stories in the different sessions are attuned on each other and attuned to the audience. Besides the content, the practical side of bringing a large group of people together needs preparation too: room reservations, parking places, printed matters being only some of the issues. Since you would like as many people as possible to show up for the sharing day, you should ensure a VISIBLE INVOLVEMENT of a company and communicate effectively using a CENTRALIZED INFORMATION POINT. While attending the sessions of the SHARING DAY, participants may lose interest as the day goes on. So KEEP IT FUN to help people stay involved and alert all the time. And finally, the initial enthusiasm of contributors may fade away when faced with the preparation work, so you should PROVIDE A BACK-UP to prevent the sharing day from falling apart if some of the contributors drop out right before the planned day.

A SHARING DAY is a big event and should only be organized when you want to bring a large group together, i.e. when there is diversity in knowledge and experience within the group and when there is a diversity of knowledge and experience you want them to pick up. Otherwise, it is sufficient to encourage knowledge sharing in smaller groups, e.g. by a training course or a coaching session.

The company KBC ICT organized a sharing day [3] "To WPF or not to WPF" to bring together the people from their work preparation community (about 140 people) and to share knowledge about their work preparation framework (WPF): the method, the tool and the process. The main goal of this sharing day was to share experiences, lessons learned and real cases with each other.

At ThoughtWorks [4], once or twice a year, they organize Away Days. Employees tend to be at different client sites and do not necessarily have the opportunity to share everything they have learned or to meet each other face-to-face. Away Days or Sharing Days give this opportunity.

2.3 PALLET OF ACTIVITIES

You will organize a SHARING DAY to bring people together to exchange knowledge and experience and to encourage networking. There exist several forms to organize such a session. For example, a *knowledge fair* with different booths where people can walk around to 'shop' for information, *plenary sessions* to reach a large group at once, *workshops* in smaller groups to collect and share experiences and ideas with respect to a specific topic or an informal *drink or reception* where people can meet and talk with each other. However, not every form is suitable for each purpose and as all people are different, not every form will work for everyone.

What is the most effective form for the SHARING DAY?

The first purpose of SHARING DAY is exchanging knowledge. However, there are different kinds of knowledge. There is optional knowledge that is nice to know for the participants and from which they can choose whether they feel there is a need, but there is also knowledge that is part of the goals of the SHARING DAY, and you definitely want people to know at the end of the SHARING DAY. Also, the information can have a general character and be meant for a broad audience. Or the information can be very specific about a particular topic. Not all forms will be suitable to transfer each kind of information.

Each form brings the information in a different way. In a presentation there will be only one way of knowledge exchange namely from the presenter to the audience, while sometimes it might be advisable to have more interaction with the audience.

Frequently, knowing something is less important than knowing who knows something. Especially in large company you can not expect everyone to share everything they know but knowing who does know is more useful. Therefore the other and maybe even more important purpose of SHARING DAY is networking. Not every form of SHARING DAY lends itself for networking.

Therefore,
Match the form of the SHARING DAY to the kinds of knowledge and purposes you wish to share.

In a *plenary session* the whole group is reached at once. Besides presentations about more general topics, these plenary sessions are also ideal to start and end the day with. An overview of the day (what can the participants expect), who has contributed to the sharing day and of course, also practical things as room allocation, timings etc. can be provided in a plenary session.

A *workshop* might be more appropriate when interaction with the audience is required. For more theoretical sessions, where you want to teach the participants a new part of the methodology or how to use a tool, you need smaller groups to be sure you can pass the message. These workshops will be more concrete than the general plenary sessions. Consequently, the danger exists that the target audience for the workshop is not as broad as the target audience for the SHARING DAY. To avoid this, let the participants choose which workshops they want to attend.

The *knowledge fair* can be used to provide information that is 'nice to know' rather than 'need to know'. People can shop for the information they want by visiting several booths. These booths can consist of real cases, the education possibilities, contact persons, etc.

Providing information is only one of the goals of a SHARING DAY. Even more important is the networking aspect. In order to encourage this networking, you need enough time and possibilities for the participants to meet and talk with each other. More informal sessions as lunch together or a *reception* at the end of the sharing day can provide this.

Within one functional domain there are different roles with different competences and knowledge (business analysts, systems analysts, technical analysts, etc.). The chosen pallet of sessions should be balanced across general sessions aiming at networking and more specific sessions offering tailored information for a particular target role or target domain.

The alternation in styles and forms has an advantage that it keeps the participants alert and interested. KEEP IT FUN is another way to achieve this.

Depending on how broad the audience for the SHARING DAY is, not all sessions might be of interest for everyone. One option is to let the participants choose which of the sessions they follow. Another option is to make different tracks where all sessions are mandatory. This last option might be less complex to organize: participants are divided in several groups and each group has to follow a prescribed path through the different sessions.

"To WPF or not to WPF" started with coffee and each participant received a documentation file (which booths at the knowledge fair, which workshops). The plenary session opened with the results the knowledge management team from the WP community achieved the past year. After the plenary session, the participants could choose 2 out of 5 workshops to attend. There was among others a workshop about the new community portal and one about the modeling tool Mega and how this is related with WP. After a lunch, the participants were invited for the knowledge fair with 11 booths (KM instruments, best practices: how and where to find them, education possibilities, real cases,…). After another plenary session with a guest speaker, the day ended with a reception.

2.4 DRY RUN

You organize a SHARING DAY consisting of several sessions.

You want to bring a clear message with the sessions on a SHARING DAY; the participants should have these right messages at a single glance.

A lot of people cooperate to organize a big event as a SHARING DAY. The different sessions, workshops and booths in the knowledge fair are prepared by several people, with different backgrounds, opinions and visions. Varying personal visions should not dominate the agreed shared overall message you want to have for the SHARING DAY.

One of the aspects of a SHARING DAY is to let people bring their own story and experiences. However, these people are not always the most experienced speakers and might be not be so confident to speak in front of a group or to lead a workshop.

Therefore,
Do a DRY RUN of the different sessions to see whether the material in the different sessions is attuned.

The first purpose of the DRY RUN is to see whether all the material presented at the SHARING DAY is attuned: attuned with each other (e.g. is there overlap between the sessions? are there contradicting messages?) and attuned with the target audience (e.g. is the material clear for the target audience? Will they need more information than delivered in the sessions?). The SHARING DAY has one general message and this should be visible in all the material. With this DRY RUN, the organizers can check whether they'll reach their goals.

Next to attuning the material, this dry run also gives confidence to the people contributing to the SHARING DAY. The less experienced speakers get a chance to practice the story they want to tell. The booth keepers get feedback whether their posters and material bring a visible message. The speakers of the plenary session might not need a test run of their presentation; often these are the experts in their field and are confident enough about their story. Also in the one way interaction of such a presentation, there is less chance that hard questions will interrupt the session compared to an interactive workshop where the audience steers the session. Nevertheless, it might still be interesting to go through the slides to see whether the story is concrete enough and the 'expert' language is adapted to the audience.

At this DRY RUN, the flyer with the core message of each session given to all participants is screened.

However, it is not the intention of the DRY RUN to perform a complete test run of all the sessions, as this might be too time-consuming. In any case, organizing a DRY RUN will add an extra cost.

2.5 VISIBLE INVOLVEMENT

You are organizing a SHARING DAY.

How to convince people to join the SHARING DAY?

Attending the SHARING DAY should be mandatory for all members of the target audience or at least strongly encouraged. However, the time pressure for other projects might be a reason to be unable to attend the event.

Therefore,
Make visible that the company attaches great importance to the SHARING DAY and stress the personal benefit of attending the SHARING DAY.

Management empowerment is an important fact in a company with a hierarchical structure. The influence of people in authority in this hierarchical structure should not be underestimated. Inviting people from management gives a sign about the importance to attend the SHARING DAY. Also mentioning the number of budget assigned to the event (for example in the invitation letter) helps to convince them of the importance to attend these sessions.

Although the main goal of the SHARING DAY is a company goal, namely to create a network for knowledge sharing, one should also pay attention to the personal interests of people. When people can see an added value in attending the SHARING DAY for their own benefit, interest and job, they will be more inclined and motivated to attend the SHARING DAY than when they have the feeling there is only a benefit for the company.

2.6 CENTRALIZED INFORMATION POINT

You want to organize a SHARING DAY. Such a day is not only created for but also by a community. From the start, a lot of communication about this event needs to be sent to the participants.

How to conserve the coherence in all the messaging about the SHARING DAY?

A sharing day is a large event that needs a long time of preparation and is gradually built. It starts with an idea, searching for interested people to cooperate, the registration, publishing the program of the day. When all this communication goes through the mailbox of the whole community, they will be swamped with emails.

Therefore,
Centralize all the communication in one point, for example the portal of the community.

The SHARING DAY is organized for a particular target audience. Often such a community has its own portal with all kinds of information. This portal can be used to centralize all the communication for the SHARING DAY. As this event needs a lot of preparation and some of this preparation needs input from the community itself, by keeping all information centralized, as the event is built up step by step, the communication can be given without losing the coherence between the different steps.

All communication for the WPF-SHARING DAY went through the portal of the work preparation community.

2.7 KEEP IT FUN

You are organizing a SHARING DAY.

How do you keep the audience alert?

Attending the several sessions of the SHARING DAY, it might be a long day to stay alert all the time.

Therefore,
Insert some frivolous elements.

In a SHARING DAY, you need to care for alternation, dynamics and iterations. Fun is a way to insert alternation and dynamics into your event which will preserve the energy. By breaking out of the normal course of the day, you keep the people alert and interested. Additionally, when you wrap the message in a nice/funny package, people will remember it longer.
However, do not go too extreme with this. Keep your target audience in mind so that the fun parts don't come across too childish.

During the WPF-SHARING DAY, participants indicated their opinion about several statements on large thermometers placed outside the auditorium. Additional, instead of normal name cards, participants wore a card with a statement about WPF.

2.8 PROVIDE A BACK-UP

You organize a SHARING DAY. Part of this SHARING DAY is organized by volunteers, for example the booths at the knowledge fair.

However, people often underestimate the effort it needs to develop an idea to a session or booth.

Lots of people will be enthusiastic at the start and come up with nice ideas to bring on the SHARING DAY. But it takes time and effort to evolve from an idea into a well-organized workshop or a completely worked-out booth for the knowledge fair. Often people underestimate this and drop out near the end of the preparation phase and close to the day of the event.

You bring one story at the SHARING DAY and all the messages in the different sessions build up to this one story and message. When people drop out, there is a risk that your concept of the sharing day falls apart.

Therefore,
Always provide some back-up material in case people drop out at the end.

At the start, while brainstorming for ideas, plenty of ideas for sessions and booths will come up of which only a few will be selected to work out. However, the other ideas can serve as back-up material. Be aware, as an organizer, to incorporate the required time to work out these ideas last minute.

Acknowledgements

The authors would like to thank Jason Yip for his very useful remarks during the shepherding process and also the participants of the 'Collaboration and Management' Writers Workshop. This paper has been written as part of the KBC-research chair on 'Managing efficiency aspects of software factory systems' sponsored by KBC Global Services NV.

References

[1] Kelly Burke and Laku Chidambaram, 'Do Mediated Contexts Differ in Information Richness? A Comparison of Collocated and Dispersed Meetings' Proceedings of the 29th Annual Hawaii International Conference on System Sciences - 1996

[2] Mary Lynn Manns and Linda Rising, 'Fearless change: patterns for introducing new ideas', Addison-Wesley, 2004

[3] The instrument SHARING DAY was developed and implemented by DNV CIBIT in cooperation with KBC ICT/ knowledge management team

[4] Direct communication with Jason Yip, ThoughtWorks

Business Patterns for Product Development (EuroPLoP 2008)

Allan Kelly - http://www.allankelly.net

1 Abstract

This paper introduces four patterns for use by software development companies, predominantly independent software vendors (ISVs), for growing their business and creating new products. Starting with a single product company these patterns describe how product and services are added to the market offering to create a whole product and then a company with a product portfolio. The product roadmap is introduced as a planning tool for product growth and enhancement.

The patterns presented here are:

- SINGLE PRODUCT COMPANY – When time and resources are scarce, focus all your attention on developing and marketing one product.
- WHOLE PRODUCT– Provide additional products and services so customers are able to recognize the promised value from the product.
- PRODUCT PORTFOLIO – Managing your products as a portfolio.
- PRODUCT ROADMAP– Create a product roadmap to show a vision for the future.

2 Audience

These patterns are intended to codify several common business practices in a pattern language so that they may be better understood, communicated and studied. Within existing companies many of these patterns already exist, albeit as tacit knowledge or embedded in operating practices.

The patterns given here are intended for those creating and applying corporate strategies. This group includes, existing managers, future managers and entrepreneurs as well as those studying to take on such roles.

In particular it is hoped that those on the receiving end of such strategies and tactics will find these patterns informative and useful. Understanding what a company is attempting, why it is acting and the implications can be benefit everyone in the organization.

Proceedings of the 13th European Conference on Pattern Languages of Programs (EuroPLoP 2008), edited by Till Schümmer and Allan Kelly, ISSN 1613-0073 <issn-1613-0073.html>.

The patterns in this paper, and others in the series (Kelly 2005a, b, 2006, 2007a, b) may be read and applied outside the domain of software companies. They may be applied to technology companies in general and to non-technology companies in some instances. The author has chosen to confine the domain and context of these patterns to software companies for two reasons. Firstly this is the domain the author knows and has experience in. Secondly, limiting the domain helps maintain the brevity of the patterns. Despite these deliberate limitations the author believes many of these patterns may be applied in contexts outside the software domain.

In parts the patterns draw on existing research and literature. Inevitably these patterns represent the author's understanding and views on how companies should go about tackling the problems identified. While there are no right answers to these problems - indeed some out dispute the problems identified – it is hoped that these patterns can help expand the understanding of business strategy in the technology domain.

3 The Patterns

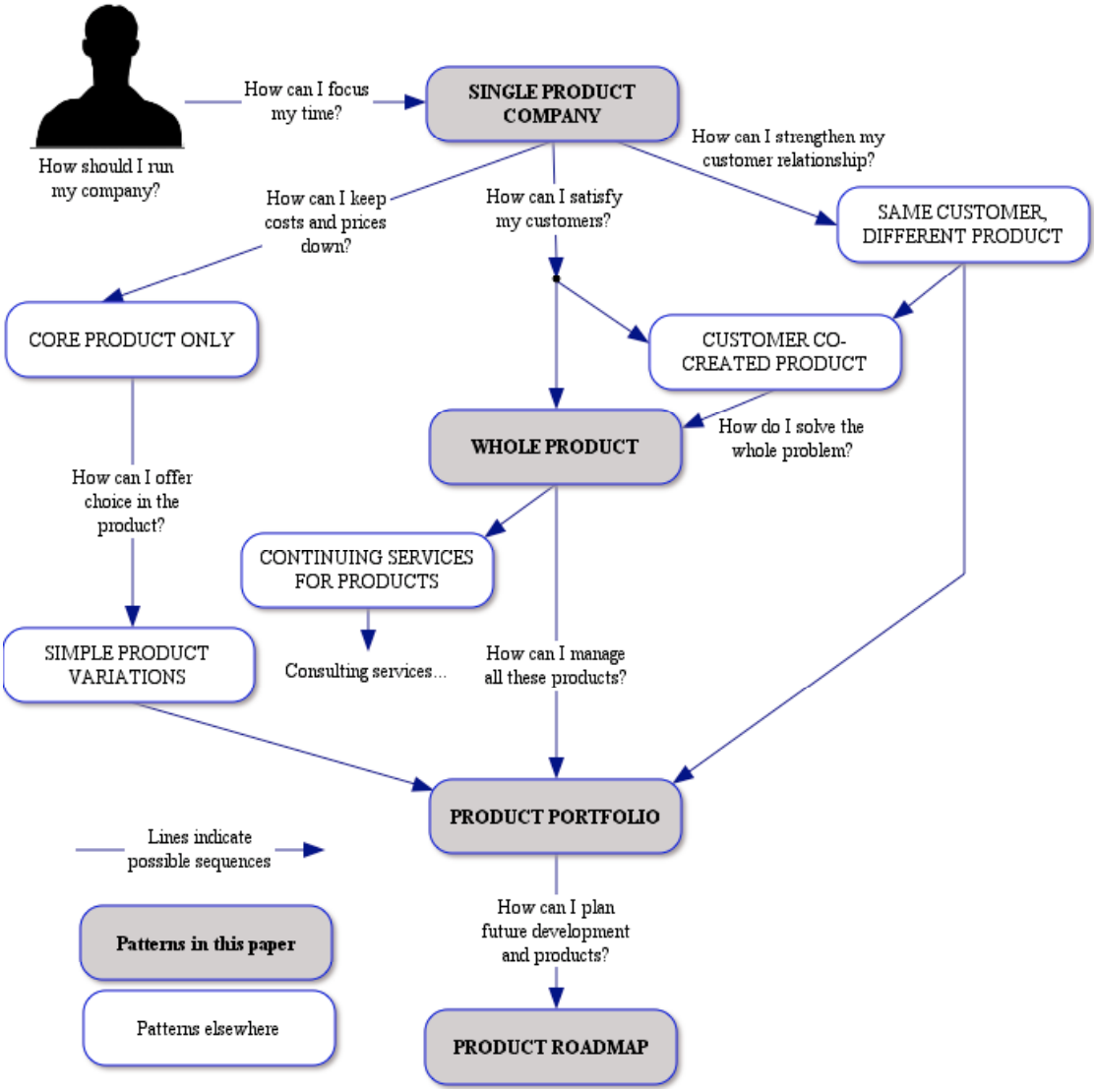


Figure 1 - Pattern sequence

SINGLE PRODUCT COMPANY Page 4	When a company is starting there are so many things to do. Focus all your attention on a single product. Get this product right and get it selling before moving onto other products.
WHOLE PRODUCT Page 8	Customers buy your product to solve a problem but solving the whole problem requires more than just your product. Therefore sell your product with everything needed to solve the whole problem.
PRODUCT PORTFOLIO Page 15	As a company grows it will offer more and more products but this makes it difficult to focus. Consider all your products as a portfolio. Balance the portfolio to achieve your corporate objectives.
PRODUCT ROADMAP Page 16	Customer and co-workers need to know what will be in future versions of the product. But you need to be able to change what features are included and when. So create a roadmap that shows future features with approximate dates. Use the roadmap to solicit views and revise the roadmap. Keep the roadmap as a living document.
CORE PRODUCT ONLY (Kelly 2005a)	Reduce costs by only supplying the core product, anything extra should be billed separately.
SIMPLE PRODUCT VARIATIONS (Kelly 2005a)	Product variations allow you to differentiate your product from competitors and provide your customers with a choice they value. But variations can be expensive to produce and support; therefore, offer simple variations on the product, e.g. choice of colours.
CONTINUING SERVICES FOR PRODUCT (Kelly 2005b)	Complex products often require continuing maintenance and support. The company that makes the product already knows a lot about it, and so is well placed to perform this activity too. By sharing knowledge between services and products operations, both can be improved.
CUSTOMER PO-CREATED PRODUCT (Kelly 2007b)	Ensure your product will do what your customers want by enrolling customers in your development process. This gives them an opportunity to influence the product design and implementation.
SAME CUSTOMER, DIFFERENT PRODUCT (Kelly 2007b)	It is easier to sell to existing customer than it is to find and sell to new customers. Therefore have additional products you can sell to your existing customers.

3.1 SINGLE PRODUCT COMPANY

Figure 2 - 1908 Model-T Ford

After several iterations Henry Ford finally found a commercial success with the Model-T Ford. The Ford Motor Company initially focused on just producing this one car and, famously, in one colour only: Black.

Context You have identified a need in the market and have decided to start a company to address the need. You believe the need is best satisfied by a product rather than a service, so will not use SERVICES BEFORE PRODUCT (Kelly 2005b).

Problem **When you start a product company what do you do first?**

Forces When you start a company the world is our oyster. There are countless opportunities for new products and vast untapped markets.

Companies are normally brought into being to do something specific. To address need in the market, an opportunity with a specific customer or exploit a new technology. But there are many ways you can go about addressing *something*, it is hard to know where to start, and even harder to know whether you are addressing it in the right way.

There are many things a new company has to do: legal status, accounts systems, recruitment, customer accounts, etc. etc. but the company founders only have so much time, energy and money to devote to all these issues. All companies have limited resources and new companies are more limited than most.

Solution **Decide on one product and focus all your attention on developing the product, delivering the product and marketing the one product.** Bring this product to market as quickly as possible.

Ask the question: "What is stopping us from delivering this product tomorrow?" Direct your time, energy and money at resolving the issues identified by this question.

Identify your target market and target customers as early as possible. Engage with them before the product is finished, they may be happy to offer advice, to beta-test the product, or act as a lead customer -

see CUSTOMER CO-CREATED PRODUCT (Kelly 2007b).

Ensure your target customers can pay for the product one way or another. You may sell the product in a single transaction, or charge a monthly fee or offer your product for free and sell advertising around it. However you decide to monetarise your product make sure the revenue will cover your costs and give enough profit to justify the investment.

When the chosen market is large, and the problems to be solved are many, then define your own niche to improve your focus. Deliberately put some opportunities out of bounds. By defining the part of the market you will address you will improve your own focus and reduce the amount of time you need to deliver a product. Having a clear idea of who your target market is, and what you have to offer should make it clearer to communicate your marketing message.

Focus on the core problem the product is solving, limit extra activity and work on both the product and the company as a whole. Leave extra functionality out of the product and limit the growth of the company organization. Defer non-essential activities like setting up a human resources department, or look to do them differently, maybe rent services or outsource work.

While developing your product avoid the distraction of offering services, avoid the temptation to develop additional products. Focus on your market, focus on the product you are developing, focus on your potential customers.

When your product is available this advice no longer holds. You may need to supplement your offering with services or additional products as described in WHOLE PRODUCT and from there to PRODUCT PORTFOLIO.

There is no guarantee that your first product will be the right one. While researching the market, building the product or even after product launch you might identify a more interesting prospect and decide to change focus. Consider using EXPEDITIONARY MARKETING (Kelly 2004), to help refine your product ideas.

Once established most companies relax their focus on the single product. They may add additional products and services around the original product (as in WHOLE PRODUCT) or they may diversify with new products – see SAME CUSTOMER, DIFFERENT PRODUCT (Kelly 2007b) and PRODUCT PORTFOLIO.

Consequences Knowing what single product you are producing, and what single problem you are solving will make it easier to focus your resources and limit distractions.

By keeping features and functionality to a minimum you can reduce development costs and shorten the development cycle. However this means the features you do implement need to be the right ones to make your product attractive.

Focusing on the product will detract from building the company organization and infrastructure. In the short term the company needs the product – and the resulting revenue – more than it needs the infrastructure of human resource departments, public relations and such. But such capabilities too long can have negative consequences. Companies may work sub-optimally if new business units and functions are not created when they are needed. For example, a dedicated technical support desk may be a distraction at the start but when there are many customers it is more disturbing to have engineers answer queries directly.

Similarly, once the product is established and revenues are flowing companies need to consider supplementary and additional products. Delaying new products may leave opportunities for competitors to enter the market. Use Whole Product and Product Portfolio, and product services like PRODUCTS WITH SERVICES (Kelly 2006), START-UP SERVICES FOR PRODUCTS and CONTINUING PRODUCTS FOR SERVICES (Kelly 2005b).

Marketing is easier when your (new) company name is associated with one product, e.g. Hoover in Europe and Q-Tips in the USA.

Companies which do not expand their product portfolio are often acquired by large companies where they form part of a portfolio.

Variations

Companies that do not follow this strategy from the beginning can still adopt this strategy later. In order to create focus shed additional products, withdraw from non-core markets and decline customer business outside the core area. Review employee incentives to ensure everyone is focused on the same thing. It is no use focusing on quality if engineers are still given bonuses for solely making delivery dates.

Focus need not be product related, although for software companies it usually is. Companies may focus instead on particular customer and their needs, or specialist activities.

In some markets it customers may expect to buy a set of similar products. For example, a customer buying a lipstick may expect to buy matching nail-tarnish, if they cannot then they may by nothing. Generally this is not the case for software companies.

Examples

Most start-up companies pass through a single product period in their early days. Originally Apple only sold the Apple II computer while Intuit started with Quicken alone.

Henry Ford is famous for offering his customers “Any colour they like so long as it is black”. Early Ford operations were totally integrated and focused on producing one car in one colour. This helped Ford to enter and dominate the early motor business but also provided opportunities for competitors. However this strength was also Ford’s weakness. By offering customers choice in the product General Motors was able to compete with Ford.

Also known as	-
Related work & Sources	CORE PRODUCT ONLY and SIMPLE PRODUCT VARIATIONS (Kelly 2005a) describe how to manage costs by focusing on a single product and how increase revenue with additional sales or variations.

3.2 WHOLE PRODUCT

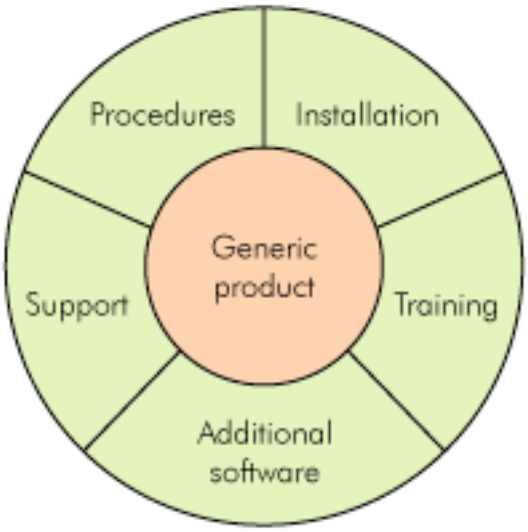


Figure 3 - The whole product doughnut

In the 1980's Silicon Graphics (SGI) set out to target the Hollywood post-production film editing process. Rather than emphasis the raw power and all-round capabilities of their machines SGI specifically presented their machines as video image editors. They added features such as video device interface ports to their machines to make their product superior to competitors for this specific task.

Context Your first products are in the market and have sold well to early adaptors. There is more to using the product that *plug-and-play*.

Problem **How do you ensure technical products deliver value to customers?**

Forces New technology is cool in its own right, it may be used in many ways and produce many benefits. But using the technology to address real world problems requires work. Some customers (early adopters) are prepared to buy the core technology and make it work for them. But many more potential customers are not prepared, or able, to put in this effort so will not buy your technology.

You wish to expand your market beyond the technical savvy early adaptors, but your product is complicated, those individuals or organizations without technical know how will find it difficult to recognize value from the product.

The value of your product can only be recognised when it is used in conjunction with specific hardware or additional software; when people are trained in the system, when the product is integrated with existing systems, when processes are changed. Each of these additional requirements put obstacles in the way of sales and customers seeing the full value of your product.

Competitors offer products that can match, or even better, your products in many tasks. Your competitors may even have advantages over you, they may have a respected brand name or their technology may be better.

Some markets may not be aware of how your technology can improve their work. Even if they are aware they need to bring together and integrate several technologies to realise the benefits.

Solution

Match your technology to your market, identify a market where your technology can deliver benefits to customers and seek to solve customer problems completely. To do this bundle your generic product with any additional products and services are needed for customers to recognize the promised value from the product. Market the complete package to customers in your chosen market.

Continually seek to identify the obstacles that stop customers from maximising their benefit from the product. Remove each obstacle either by changing the product or supplementing the offering with extra products or services.

Differentiate yourself from technology peers by addressing the needs of a specific market. If the product needs additional hardware then bundle the product with the necessary hardware. If additional software is needed to interface with other systems then supply the software. If integration services, training or support are needed then supply these – use CONTINUING PRODUCTS FOR SERVICES. You may choose to supply these products and services yourself or you may enter into partnerships with others who can supply them.

Direct your marketing effort at your chosen market. Advertise to this industry, attend their shows, sponsor their industry awards – be part of that industry. Show how your technology is better than competitors because you cater for the market needs and solve their problems.

Be clear about the problems your product solves, before adding or changing anything about the product ask: *Will this help solve the key problem?* Rather than think of your product as a set of features think of it as a single solution.

Creating a whole product and matching customer needs is not about selling more accessories and services around your product, it is about making sure your solution solves a customers problem and the customer recognises value from your product. Taking features out of your product may help the customer reach these objectives too. For example, a product cluttered with features for customers outside the core market may make the interface or installation more complicated

Consequences

Your technology is applied to a specific problem in a specific market – sometimes called a *market vertical*. The core technology product

is augmented in such a way that it fixes this problem perfectly, you sell a *whole product,* not a technology.

The technology is hidden, removed and sometimes simplified so that the product application represents a complete solution. The benefits of this solution are available far beyond the technology enthusiasts.

The value of your whole product is clearly spelt out by defining the tasks it will perform and the benefits it will provide. Focusing on the benefits and final product you actively set out to remove all barriers between customers buying the product and seeing the benefits.

You will differentiate yourself from competitors with similar technology who do not focus on your chosen market. You will be able to point to similar technology from competitors and explain why they cannot deliver the benefits you do.

Being the first to bring new technology to a specific market will give you a head start – so called first mover advantage. It will also give you a chance to define the market and product offering. Even if you do not have first mover advantage in the market, or with the technology, you can still define a niche were you will serve customers better than any competitors.

Choosing your niche, and focusing your technology into a product for a specific market vertical limits the size of your market. Instead of targeting a wide and shallow market you are aiming narrow and deep. Once you have dominated one vertical market you can repeat the exercise in another market adjacent to the first one. It is easier to tackle one vertical at a time than attack on a broad front.

As your product offering grows you will be able to justify a higher price. Fixing a specific problem in a market will help identify the value, and thus price, of your product. When your product offering contains an ongoing element (e.g. technical support or operations management) you will be able to charge regular fees. The fees are not only an additional source of revenue they are more predictable. Such fees will add to your company value because they are considered ‘high quality’.

Focusing on a specific market or market segment will mean passing over sales prospects in other area. This is necessary to create true focus but may lead to some difficult decisions, particularly when the sale in prospect is big. Continue to chase deals outside your core market will dilute the focus.

An established company adopting a whole product strategy will need change its own structure and organization. People working to support non-core markets may need to be redeployed or even laid-off. Existing customer in non-core markets also present a problem, whether it is better to continue supporting them or withdraw your product will depend on your exact relationship with the customers.

Providing services as part of a product offering can cause conflicts in product development and quality management. See the

	discussion in CONTINUING SERVICES FOR PRODUCTS (Kelly 2005b).
Variations	It is preferable to stop selling the generic product on its own. This will help focus your marketing message, simplify pricing and remove the danger of competing with yourself. But withdrawing from markets, or not allowing customers to mix and match your product with other "best of breed" may harm some of your prospects.
Examples	See *Crossing the Chasm* (Moore 1999) for a longer discussion of whole product strategy and numerous examples including: Silicon Graphics, Intuit and Documentum.
Also known as	-
Related work & Sources	A WHOLE PRODUCT strategy is the opposite of a CORE PRODUCT ONLY (Kelly 2005a) approach. Both strategies may lead to SIMPLER PRODUCT (Kelly 2007b). *Lean Solutions* (Womack and Jones 2005) advises suppliers to "Solve my problem completely." That is, provide solutions to customers entire problem not part of the problem. Such an approach would naturally lead to a WHOLE PRODUCT strategy.

3.3 PRODUCT PORTFOLIO

Figure 4 - Nokia Phones

Few large companies sell just one product, Nokia sell a range of phones to suit all tastes and budgets. You can choose between a small 6300, a large N95 with a hard disk or a Blackberry like E61.

Context You have successfully used SINGLE PRODUCT COMPANY and WHOLE PRODUCT. It is time to grow the company and you are building SAME CUSTOMER, DIFFERENT PRODUCT.

Problem **How do you decide which products to continue selling, which to introduce and which to discontinue?**

Forces Company strategy is no longer synonymous with one product. Your new strategy needs to encompass multiple products. But your resources are still limited and your potential products all demand resources. Some trade-offs and compromises are necessary but nobody wants to loose resources.

New products need time to demonstrate significant sales but demand development resources. Old products might be profitable but they are near the end of their life and vulnerable to competition. Even if you do not wish to grow the company you still need to consider new products. Customer needs and tastes change over time. Products age in the market – competitors enter and new technology change production options. Introducing new products, changing existing products and retiring old products all takes time, money and effort. You need to decide how to allocate your resources between these activities and how to reduce risks.

Being a single product company has brought you success but you now need to grow the company. Maintaining focus on one product

brought success but, by definition, you cannot focus on too many things.

What is in the best interests of the portfolio may not be in the best interest of a specific product, and vice versa. Tension arises between products serving one market and products serving another market. More tension will arise because products are build by different teams, decisions about individuals effect the portfolio and vice versa.

By following WHOLE PRODUCT you now have a cluster of product around the original product. But as this cluster grows, and as more different products are added it is difficult to see the common elements. Each additional product – or service – requires management attention; the same managers have more work to do.

Solution

Instead of managing each product as a single product manage the collective product portfolio. Delegate management of individual products to specific teams and managers. Each *product line* can then focus on its product(s) while company management should then focus on the overall portfolio of products.

This is easier said than done. There are many criteria and conflicts to manage. The portfolio needs to balance the need for an orderly introduction of new products and retirement of old products, and balance the customers need for a range of products to choose from and switch between.

There are many criteria that may be used in creating a product portfolio so it pays to define your criteria before evaluating the portfolio. Criteria need to be based on company goals and objectives, risk aversion, approach to innovation, cost of producing new products, customer need and many other factors.

With the right criteria in place the portfolio will reflect company strategy. If you are driven by near term profits then all your resources should be put into the most profitable products. Conversely, if you are looking for growth you may tolerate loss-making products that may bring in new customers and growth in the medium term.

For some companies the customer's need for a selection of products will dominate, for example Nokia's mobile phone range. Or companies may offer customers a range of products for the different stages of their lives, so Ford Europe offers the small Fiesta car for young drivers, the Focus for couples with young children and the Galaxy families.

Other companies may need to balance aging products against new introductions will be paramount. Stability and continued support may be important in some sectors while innovation and fresh products are important elsewhere.

Your product portfolio needs a unifying theme, or *core competency*. The theme may stem from your capability with some technology, or from servicing a particular type of customer, or solving a certain type of problem.

When constructing the portfolio consider how customers needs differ, why they might change, what they will need (or want) next and provide a product. By covering a variety of positions you have multiple products to offer your customers as their needs develop and change.

Both the criteria used to evaluate products and the methods used to manage the portfolio are large topics and outside the scope of this pattern. Two approaches to constructing a portfolio are detailed in the side-boxes *Vertical and horizontal portfolios* and *BCG Product Growth Matrix*. Many other approaches are possible.

Consequences

Setting portfolio criteria and positioning products against each other will highlight product priorities and inform resources allocation. The portfolio view will help balance product introduction and retirement by showing product lifecycles. This in turn will help reduce risk.

Studying the portfolio as a whole will help identify gaps and opportunities for new products. You will also see product overlap where one product is stealing sales from another. You can also see products that are in decline; these may be revived by further investment or milked for further sales without investment.

Introducing new products can offset slowing sales of an aging product. Alternatively you may be able to rejuvenate the aging product by offering it into a different market segment.

By offering a selection of products you can retain your hard won customers as the look for new and different products. When you have a relationship with your customers they will want to do business with you again; and when you sell them more products you will strengthen the relationship.

Company strategy is now concerned with a range of products you offer and how those products relate to one another. Each product group can continue to focus on their product while you focus on the portfolio.

Tensions between different products, and between individual products and the roadmap or company strategy, are easier to recognise even if they cannot be resolved.

Rather then focusing on a *whole product* you are looking at the *whole company*. Delegation becomes possible and individuals can be allowed to focus on individual products.

A balanced portfolio will allow you to invest in new products and take risks with the products you develop and introduce without jeopardising the security of the company. The portfolio will also

	allow for the organized retirement of older products so customers can be migrated to new products and support wound down. Without a unifying theme your portfolio, and company, will start to resemble a conglomerate. Although conglomerate's have advantages they often trade a discount when listed on a stock market. Portfolio management can lead to sub-optimal decisions for individual products. Some products may be held back for fear of damaging others or *cannibalising sales*. Competitors may be able to exploit such gaps by introducing their own products. For example, IBM initially held back the PC so as not to damage sales of mini-computers but competitors raced to enhance the capabilities of the PC.
Variations	The BCG matrix is a widely cited and critiqued example of a portfolio management technique (see sidebar).
Examples	Product portfolio abound, most large companies offer a range of products.
Also known as	-
Related work & Sources	Use Product Roadmap for each product then synchronise the roadmaps.

Vertical and horizontal portfolios

Portfolios may be vertically or horizontally, or a mix for both. In a horizontal portfolio the products fulfil a similar need. For example, Dell's laptop portfolio, is arranged horizontally. The laptops are broadly comparable but are alternatives aimed at different users: Inspiron is aimed at home users and Latitude at business users. A buyer will choose the laptop that best meets their needs but are unlikely to buy more than one.

Vertical portfolios are made up of products which link together. Buyers are likely to buy several products from the portfolio to work together. For example, IBM offers the Z Series mainframe, Figure 5**,** this runs the Z/OS operating system, on which can be run the DB2 database and on top of that Office Vision office automation software. In a vertical portfolio one product leads to the next.

A WHOLE PRODUCT strategy creates a horizontal portfolio of products that serve a specific need. The layers of the *stack* are unimportant to the final customer who wants a solution to some *problem*. Each generic product will have its own mini-portfolio of related products and services. Some of these may be products in their own right if developed right.

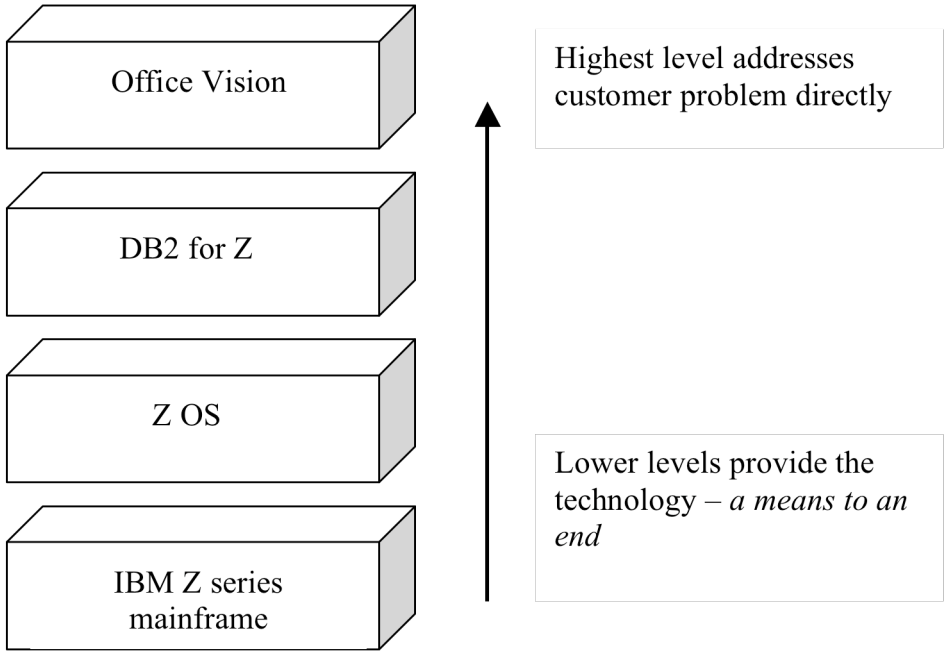


Figure 5 - IBM offers a vertical portfolio of mainframe products

BCG Product Growth Matrix

The horizontal and vertical view of the product portfolio describe how a portfolio is presented to customers. Another way of looking at a portfolio is the *Growth Share Matrix*, Figure 6, from the Boston Consulting Group (BCG). This approach looks to maximise the return for the whole company.

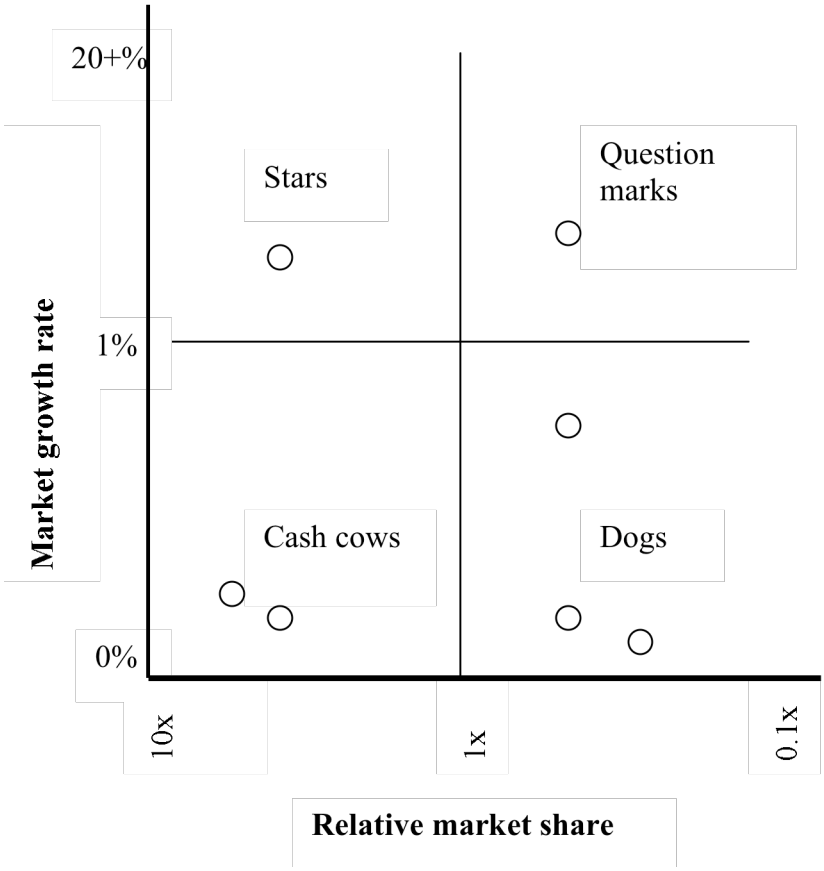


Figure 6 - Boston Consulting Group's *Growth Share Matrix*

The matrix divides products into one of four categories based on whether the company has a high or low market share, and whether the market as a whole is experiencing high or low growth.

Companies that follow the matrix are advised to maximise their returns from *Cash Cows* while minimising investments because the market is not growing. Companies are advised to discontinue *Dogs* – low sales and little potential growth - and invest in *Stars* that will produce profits in future. There is not standard advice for *Question Marks*, those products which have potential but are currently performing poor. These products require closer attention.

Such advice can be simplistic, particularly for small technology companies. More detailed analysis may consider the profitability of products and their role in providing for a *Whole Product.* An unprofitable *Dog* product may in fact be providing vital support to another product.

One problem with the growth matrix can be defining the market. *Crossing the Chasm* (Moore 1999) advise companies to define the market as narrowly as possible in order to focus action and present the company as the market

leader. Following this advice renders half the matrix pointless because we have defined our market as a market we dominate. Thus there are no *Dogs* or *Question Marks*. Conversely an expanding company may redefine its market more broadly in a search for growth. At a stroke a *Star* product can be turned into a *Dog*.

The question of market share and market definition can have a profound effect on company action. During the 1980s General Electric famously pursued a strategy of being 'number one or number two' in every market it operated in. The company exited those markets were it could not achieve first or second place market (Welch 2001). However one way to achieve leadership was to define the market narrowly. In doing so the company could miss profitable opportunities in a wider market.

3.4 PRODUCT ROADMAP

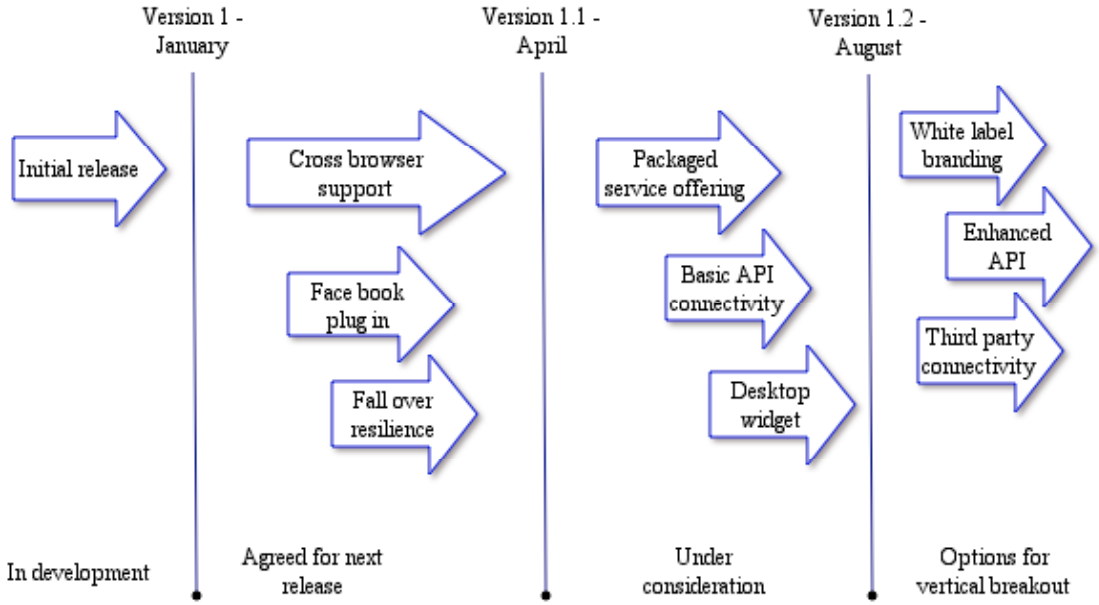


Figure 7 – Near term product roadmap

"It's tough to make predictions, especially about the future." Yogi Berra, American baseball player and philosopher

"One of the biggest roles of science fiction is to prepare people to accept the future without pain and to encourage a flexibility of mind." Arthur C. Clarke, science fiction writer

You can't predict the future for your product but you need to base your activities around a map everyone can agree on.

Context You have an existing product in the market. Now you have to tell customers and staff how the product will develop.

Problem **How do you plan for a product's future, and communicate this to customers, employees and partners when the world changes so much?**

Forces Without new products, and new versions of existing products the company will not advance. Without a vision of what will be in future products it is difficult to plan for the future and impossible for your engineers to start building.

It is always difficult to foresee the future, but lots of people want to know what your product will do in future. Your customers want to know what your product will do in future. Your organization needs to plan for the future, what resources will it need? When will it have a new product? But you cannot answer their question with absolute certainty.

Lots of disparate groups have an interest in knowing and suggesting what the product should do in future, but how do you incorporate their ideas? And how do you explain the result? The product will

need to meet future customer needs, it needs to take advantage of new technology, the product needs to fit with the company strategy and will help shape future strategy.

A plan can be formulated from these ideas and opinions but plans take time to develop and even longer to implement. During that time things change, and some things take longer than expected.

Different groups might feel the need to create different visions of the future. The Product Management group might create a roadmap which addresses customer needs while research and development create on that looks at future technology development. But multiple roadmaps will fragment your future vision, they might even conflict and may confuse customers.

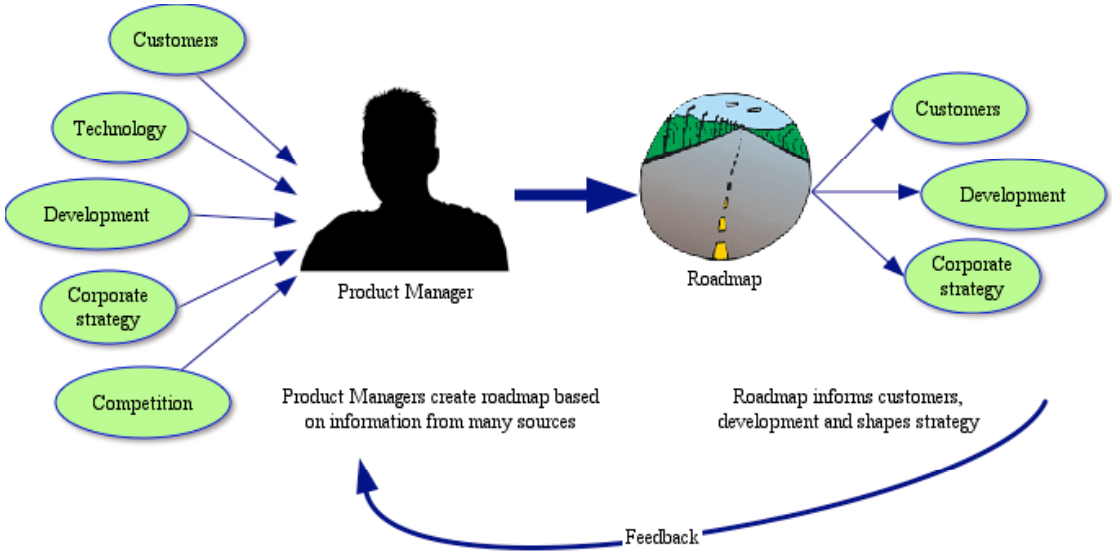


Figure 8 - Roadmap creation

Solution

Create a product roadmap to show a vision for the future. The roadmap does not contain a lot of detail, it is more about vision than execution, it shows where you are going rather than a detailed route.

Before building the roadmap decide who the stakeholders are and find out what they want from the product in the future. Customers are the most obvious (and important) stakeholders, the actual users of the product are important too – often, but not always, customers and users are the same people. People in the company will also have needs and suggestions that will be useful.

Divide the roadmap into time-buckets, perhaps by year quarters or by 'next three months, 1 year, 5 years' – whatever division works for you. Put different objectives in different buckets and keep timescales vague.

When there are lots of enhancements, features and changes to make group them into themes. Each theme should address a specific aspect of the customers problems.

Mark key events and dates on the roadmap too. For example, trade shows you wish to exhibit at, legislation changes, important financial reporting dates, anticipated competitor actions and dates important to your customers.

Show the roadmaps back to the stakeholders and listen their feedback. Present the roadmap to the whole company and listen again. Use their feedback to change the roadmap. Roadmaps are living documents and subject to change. Expect to update your roadmap at least quarterly, certainly not just for the annual report.

Incorporating different groups and listening to feedback will help to bring everyone in the company to agreement on a single roadmap. Technology development and customers needs can – and should – be shown on a single map which is simple to understand. Careful attention needs to be given to timelines so technology introductions and changes can be made.

Avoid making commitments based on the roadmap, you will need to commit to some things, for some dates, but the majority of the roadmap needs to be flexible. (Work might take longer than expected or needs may change.) The closer something is on the roadmap the more definite it is. Items that are further away (e.g. five years out) may be removed long before the date shown.

It may not be wise to show customers your full roadmap. Such a roadmap may contain information you don't want them to have, e.g. features for their competitors. You may wish to show customers versions of the roadmap which emphasis the things they are interested in, and hide other elements. Avoid these problems by never letting a sales person conduct a roadmap presentations alone.

Consequences The roadmap provides the future vision for people to work towards. It is always a best efforts map because things always change.

A roadmap describes one version of the future and provides a base for further discussion. Some psychologist call this a *transient object.* You cannot tell the future exactly but the roadmap allows you to talk about and plan.

The roadmap, the picture, accompanying documentation and verbal description are the result of many inputs. Some people will immediately see their suggestions, others will need to be shown. Some suggestions will be absent from the roadmap because it is not possible to satisfy everyone. The roadmap will allow you to explain what was left out and why. If every idea is included the roadmap will be impossibly large and complicated. What you leave out is may be more important than what you include; there are always some requests which are best turned down.

Creating a single roadmap which reconcile different departments – such as marketing and R&D – will create a unified vision across the company.

Developers can start work on developing the products and features on the roadmap. Although items on the roadmap will change there will be enough certainty at the start of the map to start work. Regular updates between roadmap owners and developers will be needed to accommodate changes on both sides.

Salespeople can use the roadmap to sell products by promising features. This can help improve sales but can be problematic when sales people sell features that are later delayed or removed.

A roadmap will raise expectations and perhaps inevitably form the basis of commitments and when a roadmap changes some commitments will be broken. Sometimes this is difficult to avoid but when commitments are being broken on a regular basis it is a sign something is wrong. Look for the underlying reason: perhaps the roadmap is being interpreted too literally, perhaps sales people are over stepping their authority, perhaps you planning and roadmap creation process needs to be improved, perhaps the development team needs more resources, or perhaps you too optimistic.

Roadmaps can be used to sow fear, uncertainty and doubt and to retain customers with promises of new features. This should not be the primary use of a roadmap.

Roadmaps show priorities so adding a new theme or feature means deciding its priority relative to other items. Conversations about the roadmap quickly turn into conversations about relative priorities – and shows there is *no free lunch*.

Variations

Examples

Also known as -

Related work & sources The author has worked with several ISV who have successfully used product roadmaps.

Creating the roadmap is a learning exercise, as it is created you will be forced to think about the future. Once created the roadmap is also a learning tool to help stakeholders learn about the future and consider options. A roadmap may be considered a scenario for the future.

Scenario planning (Schwartz 1991) is a well developed field and you might borrow some techniques. For example, try creating several roadmaps and select the most promising, Once a roadmap is selected the organization sets out to build the product described.

If you also following Product Portfolio remember to synchronise your various roadmaps where necessary. It may also be useful to produce a high-level portfolio roadmap.

Acknowledgements

Many thanks to Klaus Marquardt for shepherd this paper to EuroPLoP 2008 – one would think after two of these papers he would have had enough but he came back for a third. Thanks too to the participants of Workshop C at EuroPLoP 2008: Valerie Brown, Gwendolyn Kolfschoten, Stephan Lukosch, Lotte De Rore, Dinesha Koravangala, Birgit Gruber and Andreas Fiesser.

Figure 1 - Pattern sequence: author's own illustration.

Figure 2 - 1908 Model-T Ford: from Wikimedia, copyright expired, public domain image

Figure 3 - The whole product doughnut: Wikipedia, version 1.2, http://en.wikipedia.org/wiki/Image:Marketing-whole-product.png. Published under GNU Free documentation license

Figure 4 - Nokia Phones: Copyright Nokia 2008, taken from nokia.com press section, pictures for media use.

Figure 5 - IBM offers a vertical portfolio of mainframe products: author's own illustration.

Figure 6 - Boston Consulting Group's *Growth Share Matrix*: author's own illustration based on Wikipedia illustration, GNU Free Documentation License.

Figure 7 – Near term product roadmap: author's own illustration.

Figure 8 - Roadmap creation: author's own drawing, includes images from iStockPhoto (purchased) and Inspiration software.

History

Date	**Event**
August 2008	Workshop comments incorporated
July 2008	Workshop review at EuroPLoP 2008
March – June 2008	Shepherding revisions
January 2008	Revisions for submission to EuroPLoP 2008
December 2007	First draft

References

Kelly, A. 2004. "Business Strategy Patterns for the Innovative Company." In VikingPLoP 2004. Uppsala, Sweden.

Kelly, A. 2005a. "A few more business patterns." In EuroPLoP 2005, eds. A. Longshaw and W. Zdun. Irsee, Germany: UVK Universitassverlag Konstanz GmbH.

Kelly, A. 2005b. "Business Strategy Patterns for Technology Companies." In VikingPLoP 2005. Espoo, Finland.

Kelly, A. 2006. "Patterns for Technology Companies." In EuroPLoP, eds. L. Hvatum and W. Zdun. Irsee, Germany: UVK Universitassverlag Konstanz GmbH.

Kelly, A. 2007a. "More patterns for Technology Companies." In VikingPLoP 2007. Bergen, Norway.

Kelly, A. 2007b. "More patterns for Technology Companies." In EuroPLoP, eds. L. Hvatum and T. Schümmer. Irsee, Germany: UVK Universitassverlag Konstanz GmbH.

Moore, G.A. 1999. Crossing the Chasm. Capstone publishing.

Schwartz, P. 1991. The art of the long view. New York: Bantam Doubleday Dell.

Welch, J. 2001. Jack: what I've learned leading a great company and great people. London: Headline Book Publishing.

Womack, J.P. and D.T. Jones. 2005. Lean Solutions. London: Simon & Schuster.

Dietmar Schütz
Siemens AG, CT SE 2

Otto-Hahn-Ring 6
81739 München
Germany

eMail: dietmar.schuetz@siemens.com
Phone: +49 (89) 636-57380
Fax: +49 (89) 636-45450

BITSTREAM X-CODER

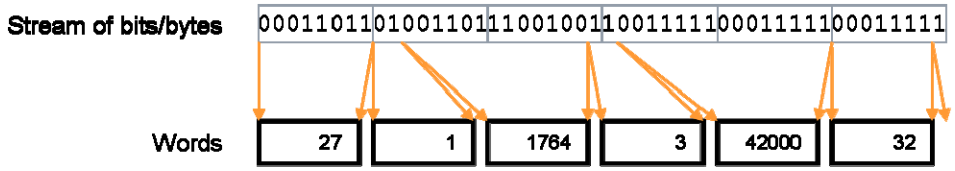
Version 1.0, EuroPLoP2008 Final

Summary This pattern provides an efficient solution for decoding densely packed bit-streams into properly aligned data structures. This conversion problem is typical for communication scenarios, where limited bandwidth and processing speed require strong optimization, and hence motivate different structures for both tasks. The proposed solution operates on bigger chunks instead of single bits, and prevalent shift operations are replaced by a finite state machine and lookup tables, thus yielding formidable throughput.

Context Software applications dealing with dense coding of information, for example communication protocols of embedded systems.

Example Consider an application scenario where wireless communication and transponders are used to exchange information with objects passing at significant speed. The contact duration is very short, and due to transmission reliability, the bandwidth is rather low too.
In order to pass as much information as possible, the telegrams transmitted are packed densely, not wasting any bits.
For efficient processing in the target environment, the telegrams need to be "unpacked" into word-aligned data structures.

Proceedings of the 13th European Conference on Pattern Languages of Programs (EuroPLoP 2008), edited by Till Schümmer and Allan Kelly, ISSN 1613-0073 <issn-1613-0073.html>.



Inflating a dense stream of bits into a word aligned data structure

The figure below is a concrete example taken from the European Train Control System (ETCS) "language" [UNISIG]. The left part shows the structure of a "static speed profile" telegram, containing two nested iterative subsections, each controlled by a counter value N_ITER. The right part shows the corresponding word-aligned data structure.

Variable	*Length [bit]*
NID_PACKET	8
Q_DIR	2
L_PACKET	13
Q_SCALE	2
D_STATIC(k)	15
V_STATIC(k)	7
Q_FRONT(k)	1
N_ITER(k)	5
NC_DIFF(k,m)	4
V_DIFF(k,m)	7
N_ITER	5

Telegram structure definition (with two nested iterative subsections)

```
struct sPACKET_27 {
  int packetId;           //NID_PACKET
  int direction;          //Q_DIR
  int packetLength;       //L_PACKET
  int scaleFactor;        //Q_SCALE;

  int nSpeedData;         //N_ITER +1
  struct sSpeedData {
    int staticDistance;   //D_STATIC
    int staticSpeed;      //V_STATIC
    int frontIndicator;   //Q_FRONT

    int nCategory;        //N_ITER(k)
    struct sCategory {
      int categoryId;     //NC_DIFF
      int cateorySpeed;   //V_DIFF
    } aCategory[33];
  } aSpeedData[33];
};
```

Corresponding target data structure definition (in C)

Another (simpler) example is base64 [Base64], a popular binary-to-text-encoding scheme used to transmit binary data across media that only support ASCII-printable characters (who's most significant bit equals zero). In this scheme, three in-bytes are inflated to four out-bytes, where every out-byte holds six of the 24 in-bits.

Problem

Processing units usually operate on aligned data (such as bytes and words) much faster compared to addressing and working on single bits, thereby dealing space for time. On the other hand, from the communication perspective, time is directly related to space. Hence, data structures are packed densely for faster transmission, and unpacked again for quicker processing. This artifice relieves the operational code from handling single bits, but nevertheless the non-productive packing routines burn CPU

resources by shifting bits back and forth. How can the packing/unpacking be done in an efficient way?

The following **forces** influence the solution:

- *Sequence of bytes*. Low level drivers usually operate on a sequence (stream or array) of bytes when transferring binary data, e.g. via a serial interface (UART). Hence, this is an appropriate representation of the "raw" bit stream.
- *Performance and Granularity*. Operating on a fine granularity (such as bits) automatically raises the number of objects to handle, with negative consequences for the performance of the overall operation. In addition, extracting bits from words requires additional effort.
- *Telegram format*. A telegram is a (nested) sequence of single elements, sometimes containing control content (e.g. length, number of consecutive elements, etc.) that have to be taken into account during processing the telegram. The single elements often adhere to a "processable" format (e.g. two's-complement), but might offend them too, especially in heterogeneous environments.
- *Telegram types*. Typically, there are various types of telegrams. Each of them comes with its own syntax, and needs to be handled accordingly.
- *Ease of code*. The telegram structure and corresponding data structures should be clearly visible in the code, not obscured by incomprehensive activities.

Solution

Although your granularity is bits, operate on words: read and convert information in chunks as big as possible, at least bytes.
To this end, transform the bit reading functionality into a finite state machine (FSM), where each state represents distinct values for the bits not processed yet in the "head" byte of the input stream. A transition from on "state" to another one processes as many bits as possible at once, and accumulate these chunks into the aspired result. Embed this state and its handling into a function that is responsible to read a single element of the telegram (unpack a short sequence of bits from the input into a word). Use this function to assemble the structure of the whole telegram.

Structure

The *Input Stream* is a densely packed sequence of bits, typically stored as a sequence of bytes or words. The first bit of the input stream is not necessarily the most significant bit (MSB) of the first word or byte. Instead, there might by some leading junk bits that must be ignored[1].

[1] This situation can arise for example when receiving data from a transponder. Typically, such messages are repeated in a cyclic manner, and the continuous data stream must be inspected first to detect a position that fits the start condition of the telegram.

The *target structure* is a memory location where the decoded telegram should be stored into. Typically for embedded environments, this is a “flat”, memory section with fixed size, and can be addressed via a structure definition. In other, less restrictive environments, tree structures build from several nodes and leafs might be an option too. All elements in the target structure are available in an optimal format: they can be used “as is” for further processing.

The core element of this pattern is the *head buffer*, a small buffer that is responsible to store the next bits of the stream for further processing. It contains an aligned snippet from the stream, as well as the information how many bits of that are remaining for processing. Both pieces are coded together in a *State* variable.

A *finite state machine* (FSM) specifies the behaviour when decoding single elements out of the stream. Input values are the actual state, and the number of bits missing to complete the current telegram element. A transition “consumes” as much bits as possible from the state, yielding the new state, the remaining length, and the decoded data, properly aligned. The output values are taken from *transition tables*, that look-up the result using indexed access based on the inputs.

Since decoding an element might require more that one transition of the FSM, an *accumulator* is necessary to store and combine the partial results until the whole element is completed.

The structural parts listed above are managed and orchestrated together into a readBits *function* that is responsible to decode an Integer (a word) by consuming a given number of bits from the input stream.

Last but not least, a structure parser is responsible to process the sequence of single elements in the input stream and extract the corresponding telegram structure in the aspired output format. This task incorporates parsing optional, iterative, and nested structures based on control data embedded in the stream. In addition, some adaptations on the data might become necessary, such as scaling of values or conversions to the “local” representation (e.g. big-endian number formats).

Dynamics

There are two important dynamic scenarios that characterize this pattern, *initialization* and *decoder loop*.

For i*nitialization*, the state as well as the input stream must be set into a defined condition. Usually, the input stream can be used as provided by the corresponding environment (either as a quasi-infinite source of bytes/words that can be read consecutively, or as a continuous memory region with fixed length). More care might be necessary on initializing the state. If the first bit of the input stream is not the MSB of the first word, the valid bits must be mapped correctly into the state variable.

The *decoder loop* covers the scenario if a single element spreads across two or more consecutive bytes/words in the input stream. Since the head buffer

can only contain data from one byte/word, it will be emptied on the first pass, and hence must be reloaded until the element is complete.

Implementation

The implementation of this pattern incorporates at least the five steps described below.

1 *Decide on the chunk size.* The chunk size should correspond to the size of an operatable unit (byte, word) of the CPU. Bigger chunk sizes yield better performance (linear ratio), but dramatically rise the size of the lookup tables. For most environments, one byte is an appropriate choice. Two bytes are an option if space is nothing to worry about. Four bytes are too much to be handled with lookup tables, thus only applicable for the calculation approach (see step 3 for details).

☛ In the example implementation, the chunk size is a byte. □

2 *HeadBuffer, coding of state.* The state must reflect the remaining valid bits of the chunk, and is stored in the head buffer.

☛ Given that the first 3 bits have been processed previously, there are 5 remaining bits in the head buffer, hence 2^5 = 32 different states are necessary to reflect the possible variants. This sums up to 511 states for zero to eight remaining bits with distinct values. □

A simple approach uses a byte variable to store the bit values, and an integer variable indicating the number of valid bits (the rightmost bits in the byte). Using both values as lookup index, the table contains n* 2^n entries, with n as the number of bits in the “chunk” sizeof(chunk)*8. Unfortunately, this wastes a lot of space, since for small numbers of remaining bits the leading bits in data are irrelevant.

A more effective approach is to combine both variables into one by using a simple coding scheme:
Bits consumed already (in the example below indicated by a dash -) are set to “0”, a single “**1**” indicates the start of the “valid” part, and is followed by the values of the valid bits.

☛ State is stored in the head buffer as an integer, coded as concatenation of '0'* + '**1**' + ”remaining bits”. The state “values” range from 1 to 511. The following table shows examples for concrete state codings.

Scenario	Logical state	Coded state (content of head buffer)
five remaining bits, values 0 1 0 1 1	`---0 1011`	`0000 0000 001`**`1`**`0 1011` = 43_{10}
no remaining bits	`---- ----`	`0000 0000 0000 000`**`1`** = 1_{10}
eight remaining bits, all 1	`1111 1111`	`0000 000`**`1`**` 1111 1111` = 511_{10}

□

3 *Transition function and lookup tables*. Consuming a specific number of bits from the head buffer causes a transition from one state to another. The maximum number of bits that can be consumed at once is limited by the number of remaining bits in the head buffer. If the buffer runs empty, the missing bits will be read later after reloading the buffer.

For each pair of (state, bitsToConsume) the subsequent state can be easily computed. The same is true for the value of the extracted bits, and the still missing bits to be read later.

For optimal performance on most processors, these functions are not computed at runtime. Instead, the results are stored in appropriate lookup tables. These tables may consist of previously calculated constants, or can be initialised at runtime.

Note: The look-up approach might be suboptimal for modern processors with data caches. In such cases, there is a high probability for cache misses when accessing the lookup tables, causing a much slower access to (uncached) memory, which results in significant performance drops. In such environments, calculating the "lookup" value at runtime can be quicker than real lookup, providing more speed and less memory consumption.

4 *Implement the bit-reading function*. The following code describes the core of the read routine.

```
readBits( byte &buf;       // pointer to input stream
          unsigned n;      // number of bits to consume
          unsigned &result // resulting value
          ) {

  static unsigned z = 0;   // „empty" byte as start state
  unsigend accu = 0;       // accumulated result

  while( n > 0 ) {  // there is something to read
    if( z == 0 )
      z = *(buf++); // „refill" state from input stream

    z_save = z;
    accu |= value(z_save, n);  // look-up (part of) value
    z = z_new(z_save, n);      // look-up new state
    n -= used( z_save, n );    // look up number of missing bits
    }

  result = accu;
  }
```

This core routine might be extended with handling of error conditions (e.g. premature end of input stream). Another add-on might be an explicit initialization of the "state" to a value different than 0, in order to cope with bit streams that do not start at a byte boundary in the input buffer.

The following show the values for `z`, n, and accu for two examples. The first example consumes only a part (4 bits) of the input stream (buffered in `z`, containing 6 bits), processed in on step. The second example consumes 11 bits, and hence needs to iterate through the loop and to refill the buffer.

```
           z n      accu                  new z    n'
-------------------------------------------------------------------

Example 1:
    --011011 4   + 0000000000000110  ------11    0
                   0000000000000110
                 = 6₁₀

Example 2:
    --011011 11    0000001101100000  --------    5      // refill z
    10011110 5   + 0000000000010011  -----110    0
                   0000001101110011
                 = 883₁₀
```

5 *Implement the structure parser*. The structure parser might be implanted in two different ways: either hard-coded or by means on an Interpreter (see [GOF94] for details).

A hard-coded parser might be more efficient, but can (depending on the size and number of telegram structures) result in a huge amount of code.

The probably smarter way is an interpreter, with data structures that controls the behaviour: a sequence of read operations processes the input stream, and writes the resulting values through a "write pointer" (base plus offset) into the target structure, while a small stack automaton keeps track of nested telegram structures.

Example Resolved

The thousands of line of code of hard coded structure parsers are obsolete, replaced by a small read function and the structure interpreter, both fitting on a single page. They are complemented by declarative code that provides the parser "programs", reflecting the telegram structures, automatically generated from XML-based structure definitions. Execution times are significantly improved, and maintenance is eased up.

Consequences

BITSTREAM X-CODER provides the **benefits** depicted below:

- *Speed.* The method described above is ten to a hundred times faster compared to a bitwise reading and recombination of the encoded values.
- *Code reduction.* Especially in conjunction with an interpreting structure parser, the required active code is rather small. In addition, it does not depend on the number and complexity of different telegram structures.

On the other hand, the pattern carries the following **liabilities**:

- *Complexity*. Less obvious than the "straight forward" implantation, not easy to understand.
- *Memory Consumption.* The transition tables are getting quite big, especially when choosing bigger chunks for processing (exponential growth in size for linear increase in speed).

Variants

The pattern can be applied "the other way around" to implement the encoder. Accordingly, the tail buffer is extended until it is full, and then flushed to the output stream.

Some compression algorithms share many commonalities with encoding and decoding bitstreams. For example, zip uses Huffman codes (the length of the "code" relates to the relative frequency of the corresponding character.

Credits

Thanks to my shepherd Peter Sommerlad, for his patience and supportive suggestions. I also thank the participants of the writers workshop at EuroPLoP2008 (André L. Santos, Carsten Hentrich, Diethelm Bienhaus, Jürgen Salecker, Paul G. Austrem) for their valuable feedback. Thanks to my working student Omar Farooq, he implemented the core of this pattern in the context of the "ETCS language" defined in [UNISIG].

References

[GOF94]
E.Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns, Elements of Reusable Object-Oriented Software; Addison-Wesley 1994

[POSA96]
F. Buschmann, R. Meunier, H. Rohnert, M. Stal, P. Sommerlad: Pattern Oriented Software Architecture, A System of Patterns; Wiley 1997

[UNISIG]
UNISIG SUBSET-026
http://www.era.europa.eu/public/Documents/ERTMS%20Documentation/Mandatory%20Specifications/SRS%20230.zip

[Base64]
Wikipedia article on Base64
http://en.wikipedia.org/wiki/Base64

Modular Hot Spots: A Pattern Language for Developing High-Level Framework Reuse Interfaces using Aspects

André L. Santos[1] and Kai Koskimies[2]

[1] Department of Informatics
Faculty of Sciences, University of Lisbon
Campo Grande, 1749-016 Lisboa
PORTUGAL
andre.santos@di.fc.ul.pt
[2] Institute of Software Systems
Tampere University of Technology
P.O.BOX 553, FIN-33101 Tampere
FINLAND
kai.koskimies@tut.fi

Abstract. Applications based on an object-oriented framework can be built by programming against the framework's reuse interface. Mastering a framework is typically a time-consuming and difficult task. This paper presents a pattern language for developing higher level reuse interfaces for an existing framework. When applying the patterns that constitute the language it is implied that the framework becomes enhanced with an additional layer of reusable modules that rely on aspect-oriented programming. These modules are referred to as Modular Hot Spots. They modularize existing hot spots, enabling a framework-based application to be built in a stepwise way and at a higher abstraction level than if using the conventional reuse interface. By raising the abstraction level, it is intended that the development of framework-based applications becomes facilitated.

Proceedings of the 13th European Conference on Pattern Languages of Programs (EuroPLoP 2008), edited by Till Schmmer and Allan Kelly, ISSN 1613-0073 <issn-1613-0073.html>.

1 Introduction

An *object-oriented framework* [4] (hereinafter, simply *framework*) embodies the abstract design and implementation of a family of related applications. Framework-based applications are developed against the framework's *reuse interface*, i.e. the classes, interfaces, and methods, which an application developer has to deal with in order to build an application. Depending on the framework nature, an application may be developed by *specialization* (*white-box* reuse) or *polymorphic composition* (*black-box* reuse). Most often, an application has to be developed using both means, given that most frameworks have a *gray-box* nature.

A *hot spot* is a fragment of the reuse interface that enables the adaptation of a certain variation point in framework-based applications. A hot spot typically involves more than one framework class, while on the other hand, a same framework class may be involved in more than one hot spot. Therefore, there is a many-to-many mapping between variation points and framework classes that support them (illustrated in Figure 1).

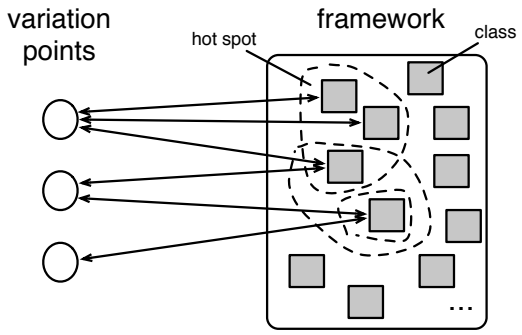


Fig. 1. Many-to-many mapping between variation points and framework classes.

The described many-to-many mapping implies that there are classes of a framework-based application that will be *tangled* with respect to the adaptation different variation points, while the adaptation of certain variation points is going to be *scattered* among more than one class of the framework-based application. *Tangling* implies that modifying the adaptation of a variation point in a framework-based application requires to cope with code statements that pertain to other variation points, whereas *scattering* implies that the modification may involve more than one class.

The work in [8] proposes a technique based on *aspect-oriented programming* (AOP) [5] for developing framework reuse interfaces using *spe-*

cialization aspects (see Figure 2). These are reusable modules that modularize hot spots and enable to build framework-based applications on a higher abstraction level than if using a conventional reuse interface. A framework-based application is implemented in *application aspects* which inherit from the specialization aspects.

This paper presents design patterns for enhancing a conventional reuse interface with specialization aspects, in the form of a pattern language that we refer to as Modular Hot Spots. The patterns can be used together for solving the problem of developing the several modules which form the higher level reuse interface. By having Modular Hot Spots it is intended that the new reuse interface has:

- *Modular reuse interface.* Framework-based applications can be developed in a stepwise way. Each application module is the adaptation of a Modular Hot Spot and implements an increment that does not require modifications or knowledge about the internals of existing modules. The capability of developing application increments without having to modify or understand existing code is beneficial with respect to evolution.

- *Less hook methods.* Through the adaptation of Modular Hot Spots, framework-based applications can be developed without dealing with as many hook methods as in conventional solutions. This contributes to have a *narrow inheritance interface*, a principle that states that only a few hook methods should be required to be given per each ap-

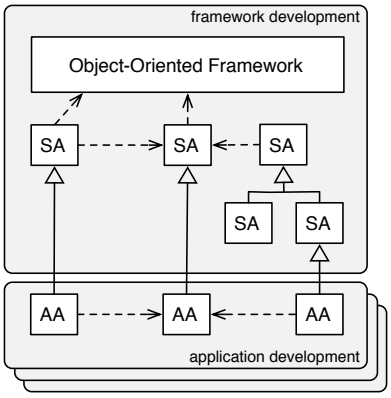


Fig. 2. Specialization aspects (SAs) and application aspects (AAs).

plication class [11].

– *Less application-relevant methods.* Framework-based applications are able to be built without using as many framework methods as in conventional solutions. This may imply that whole framework classes/interfaces will become irrelevant to applications when having the Modular Hot Spots. Having a reduction in the number of framework elements that an application developer has to deal with, reduces the size of the reuse interface, and therefore, facilitates the task of learning it.

Given the above points, the abstraction level is raised with respect to the development of framework-based applications. Frameworks tend to evolve from white-box to black-box [7]. When developing Modular Hot Spots, a framework is transformed in this direction, too. However, the framework reuse interface can become more high-level than the one of a conventional black-box framework.

The knowledge embodied in the language results from developing Modular Hot Spots using AspectJ [1], for the frameworks JHotDraw [9] and Eclipse Rich Client Platform (RCP) [6]. The former is a framework for building editors for structured graphics, while the latter is a framework for building GUI applications based on the Eclipse's dynamic plugin model and UI facilities.

The target audience of this pattern language are framework developers that seek for solutions to provide higher level reuse interfaces, enabling framework-based applications to be built more easily.

Section 2 introduces a simple framework that is used as a running example in the description of the patterns, and several scenarios where specialization aspects can be beneficial. Section 3 presents an overview of the pattern language. Section 4 presents an AspectJ idiom that is used on the implementation of the patterns. Sections 5-10 present the patterns that constitute the pattern language. Section 11 revisits the example framework taking into account the new reuse interface that resulted from all the examples given throughout the patterns. Finally, Section 12 concludes the paper.

2 Example Framework

This section introduces a simple example of a framework, which can be used to build GUI *applications*. An GUI application has *actions* that can

be triggered by the UI elements. The action can be either application-specific or provided by the framework. An application may have *menus*, which may contain submenus. The menus may contain either items that trigger application actions or other menus (i.e. the submenus). Implementation-wise there is no distinction between a menu and a submenu (i.e. they are represented by the same class). Figure 3 contains an UML class diagram depicting the classes of the framework's reuse interface (in gray), and an example application (in white) based on the given reuse interface. Below we present Java code that implements the example framework-based application.

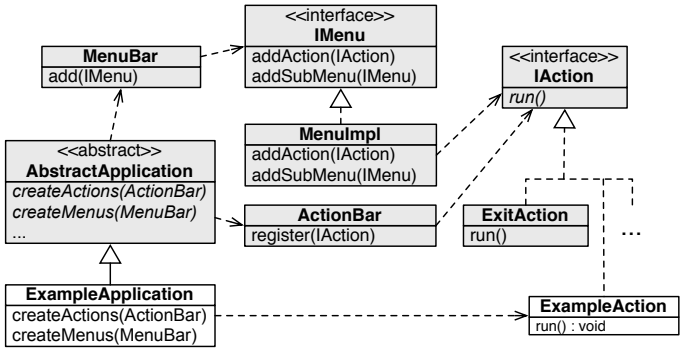


Fig. 3. Reuse interface of the example framework (gray), example application (white).

```
public class ExampleApplication extends AbstractApplication {
  private IAction myaction;
  private IAction exitaction;

  protected void createActions(ActionBar a_bar) {
    myaction = new ExampleAction();
    a_bar.register(myaction);
    exitaction = new ExitAction();
    a_bar.register(exitaction);
  }


  protected void createMenus(MenuBar m_bar) {
    IMenu menu1 = new MenuImpl("Menu1");
    menu1.addAction(exitaction);
    IMenu menu2 = new MenuImpl("Menu2");
    menu2.addAction(myaction);
    menu1.addSubMenu(menu2);
    m_bar.add(menu1);
  }
}
```

```
public class ExampleAction implements IAction {
  void run() {
    // do something
  }
}
```

The main class is a subclass of AbstractApplication. Application developers must be aware that createActions() is executed before createMenus(). The sample framework-based application has two actions, an application-specific one, ExampleAction, and the framework-provided ExitAction. It has a "Menu1", which has the exit action and a submenu "Menu2" that has the application-specific action.

Usage scenarios

The following list presents a set of scenarios where application developers (i.e. the ones who use the framework) may be faced with difficulties. Each of these scenarios is associated with a goal of application developers.

- *Scenario 1, plugging menus.* The application concept *menu* is represented directly by the interface IMenu, which MenuImpl implements. Therefore, it should be easy for an application developer to locate it. However, once the interface/class is known, it is necessary to find out how to plug the menu in the application. Given that the *application* concept is represented abstractly by the AbstractApplication class, one would go to inspect that class, and then, to realize that there is a hook method for the intended purpose (i.e. createMenus(...)). Plugging the menu involves modifying the method body, which may have existing statements. Therefore, in order to implement the goal of plugging a menu, one has to "interfere" with statements pertaining to other goals (i.e. other menus and their contents).

- *Scenario 2, menu context.* The application concept *menu* can be used in two different contexts, either as an *application* menu or as *submenu* of another menu. As explained in Scenario 1, by knowing IMenu and MenuImpl one does not know where and how the menus can be plugged. If one has an existing application menu *m1* and wants that menu to become a submenu of another menu *m2*, besides understanding the subclass of AbstractApplication (Scenario 1) to remove the statement that plugs the menu, there is need to locate where *m2* is instantiated and to know its interface to add *m1* to it. Therefore, changing the context of an existing menu requires changes both in

statements pertaining to the original context and in statements pertaining to the new context.

– *Scenario 3, associating actions.* The application concept *action* is represented by the interface IAction. Actions may be associated to *menus*. Suppose that there is an existing application with an action a and a menu m, and that one wants to associate a to m. In order to do so, one has to inspect the hook method createActions() of the subclass of AbstractApplication to find out the instance of a, and then, to modify the hook method createMenus() by finding the instance of m and adding a statement that associates a to it. Therefore, an association between two application elements involves two parts of a module (the subclass of AbstractApplication) which is not directly related to those elements.

Each of the given scenarios can be improved by applying the Modular Hot Spots pattern language. When addressing a scenario by applying a pattern, it might happen that a scenario with a new problem arises. In these cases, there are other patterns for overcoming the new problems.

3 Pattern Language Overview

Figure 4 gives an overview of Modular Hot Spots. The diagram contains related patterns and idioms represented in white, while the actual patterns/idioms of the language are represented in gray. Design patterns are represented in ellipses, whereas AspectJ idioms are represented in circles.

Hot spots based on Template Method [2] are typical starting points for applying the pattern language. It is common that an application framework applies at least one Template Method on the main class that initializes the application. A Template Method has one or more hook methods, which have to be overridden by application developers. A Composition Hook Method (Section 5) is a hook method that exposes an object instantiated by the framework as a parameter, with the purpose of enabling applications to plug objects in the exposed object. While this pattern is not related with the development of Modular Hot Spots directly, it describes a common solution that hints where it is suitable to have a Self-Pluggable Object (Section 6). As we will see, Composition Hook Methods are "predictable" and can be completed by a Self-Pluggable Object, after which the application developer no

longer has to deal with those hook methods. In the context of the example framework given in Section 2, this pattern is suitable for improving the *plugging menus* scenario.

A Self-Pluggable Object is a hot spot that enables its adaptations to localize both the creation of an object representing an application element and its composition with another application element. It may be plugged in another Self-Pluggable Object and it may have Composition Hook Methods itself. It can be implemented using a Template Pointcut (Section 4). A Template Pointcut is an AspectJ idiom that combines the idioms Abstract Pointcut and Composite Pointcut [3].

A Multi-Context Self-Pluggable Object (Section 7) is a special kind of Self-Pluggable Object that is suitable in cases when the object can be plugged in different application contexts (elements). The Multi-Context Self-Pluggable Object pattern is suitable for improving the *menu context* scenario. An Abstract Self-Pluggable

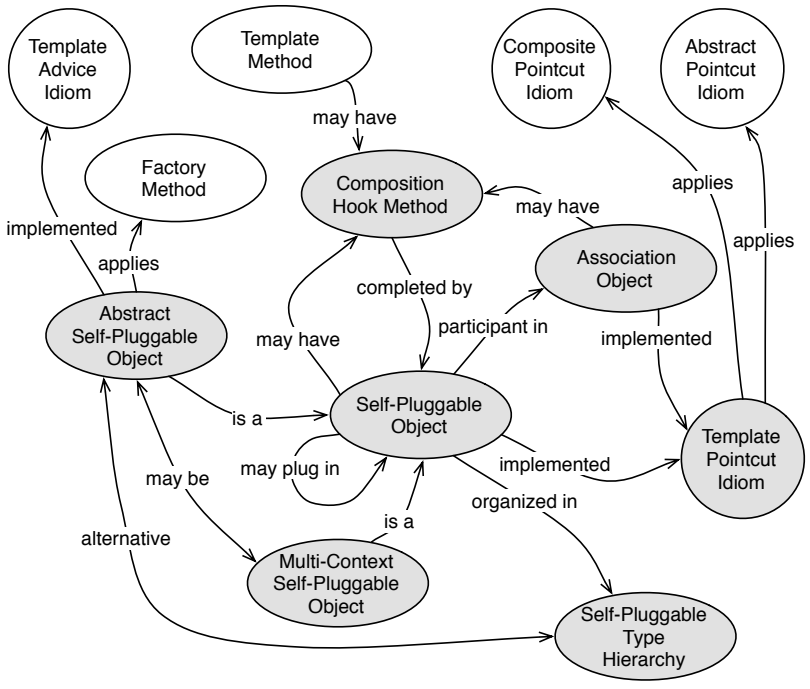


Fig. 4. Modular Hot Spots pattern language (in gray). The elements depicted in white are patterns or idioms previously described by other authors.

Object (Section 8) is a module suitable for structuring a set of related Self-Pluggable Objects, so that the behavior that plugs those objects can be reused. It applies Factory Method [2] and can be implemented using the Template Advice idiom [3]. An alternative to structure a set of related Self-Pluggable Objects is to have a Self-Pluggable Type Hierarchy (Section 9), which merges the implementation of types and the plugging of objects in the applications. The patterns Abstract Self-Pluggable Object and Self-Pluggable Type Hierarchy are two alternatives that are suitable for solving a design problem that can emerge from applying either Self-Pluggable Object or Multi-Context Self-Pluggable Object.

Finally, an Association Object (Section 10) enables to establish associations between Self-Pluggable Objects. This pattern is suitable for improving the *associating actions* scenario.

The examples of applying the patterns are given in Java, using AspectJ as the AOP language. Although the patterns were only experienced in AspectJ, they are not necessarily specific to it. An AOP language for a base object-oriented language, that features method execution pointcuts, abstract aspects, and abstract pointcuts, should be suitable for implementing the patterns. For instance, the patterns should be applicable to AspectC++ [10], AspectJ counterpart for C++.

In the figures that illustrate the solutions, the framework modules are always represented in gray, whereas the white classes represent application modules. Aspects are depicted with a class with stereotype ≪aspect≫. Pointcuts and advices are represented in the method's placeholder using the stereotypes ≪pointcut≫ and ≪advice≫, accordingly. Stereotyped dependencies represent pointcut definitions, where the stereotype name represents the pointcut name.

4 Template Pointcut: an AspectJ Idiom

This section presents an AspectJ idiom referred to as Template Pointcut. Its name results from an analogy with the Template Method pattern. In a Template Method we have a partially implemented method which uses abstract methods that are given by subclasses. In the case of a Template Pointcut, we have a partially defined pointcut within an aspect module that uses abstract pointcuts that are given by the subaspects.

Context

An aspect module transforms (by *weaving*) other modules, which can be either classes or other aspect modules. A reusable abstract aspect is a module from which other aspects inherit (the subaspects), reusing its implementation. The scope of applicability of a reusable aspect may be restricted to a certain kind of base modules. The advantage of doing so is that the reusable aspect may assume certain characteristics of the modules which are going to be transformed. For instance, the reusable aspect may be applicable to all subclasses of a certain class, an therefore the common inherited methods may be safely used by the aspect.

Problem

How to implement a reusable abstract aspect so that its advice can only take effect in a partially defined set of join points, while being able to generalize the commonalities between those join points?

Forces

- The information factored out to the reusable aspect should be maximized.
- The more "black-box" the reusable aspect is, the better.
- The simpler the pointcut definitions in the subaspects are, the better.
- The less one needs to know about the modules that an aspect transforms, the better.

Solution

Implement an abstract aspect containing a Composite Pointcut (the *template*) that is defined as the intersection of certain join points with another Abstract Pointcut (the *hook*). The advice takes effect on the Template Pointcut (Figure 5). Subaspects of the abstract aspect have to define the hook pointcut.

Example

Consider a reusable aspect that can be used to transform classes that inherit from the following abstract class.

```
public abstract class AbstractClass {
   /* ... */
   public String method();
}
```

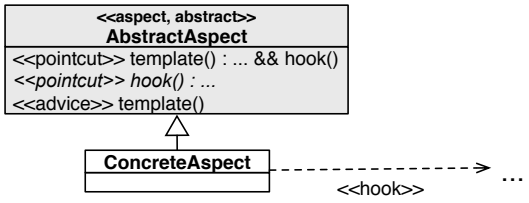


Fig. 5. Template Pointcut idiom.

The following is a reusable aspect with a Template Pointcut defined as the intersection between the execution of method() within subclasses of AbstractAspect and a hook class, which is to be given by the abstract pointcut hook(). The definition of hook() is intended to match a particular subclass of AbstractAspect, which the aspect will transform.

```
public abstract aspect AbstractAspect {
   private pointcut template() :
      within(AbstractClass+) && hook() && execution(String method());

   protected abstract pointcut hook();

   after() returning(String s) : template() {
      /* do something, e.g. */
      System.out.println(s);
   }
}
```

Although the pointcut template() is declared separately, it could be incorporated directly in the advice. Assuming the existence of a subclass of AbstractClass named SomeClass, the code below shows how the aspect could be used for activating the transformation of SomeClass.

```
public aspect ConcreteAspect extends AbstractAspect {
   protected pointcut hook() : target(SomeClass);
}
```

A Template Pointcut is particularly useful to relieve the one who reuses the aspect from understanding details of the modules where the aspect takes effect.

5 Composition Hook Method

Context

Template Method is an elementary and common pattern for enabling framework specialization, where adaptation is achieved by subclassing.

The role of the hook methods that have to be overridden is often to plug objects on the application.

Problem

How to define hook methods for the purpose of enabling object plugging, so that they are intuitive to use?

Forces

- Reuse interfaces should be as simple as possible. By reading a hook method signature, it should be intuitive what the method has to do and how.
- The less framework methods that one has to know for building an application, the better.

Solution

Define hook methods that expose in their parameters objects that are instantiated by the framework. These exposed objects are accessed by applications for composing other objects. The intent of a Composition Hook Method (Figure 6) is intuitively given by the method signature, while the way how to plug objects on the exposed object is given by its interface.

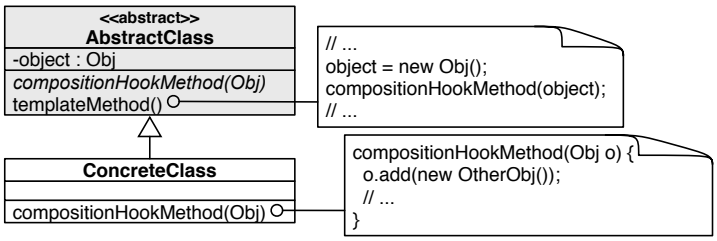


Fig. 6. Composition Hook Method pattern.

Example

Considering the example framework, the abstract class AbstractApplication could be something like shown below. The class constructor is the

template method, and there are two Composition Hook Methods for plugging actions and menus.

```
public abstract class AbstractApplication {
  private ActionBar a_bar;
  private MenuBar m_bar;

  protected AbstractApplication() {
    a_bar = new ActionBar();
    createActions(a_bar);
    m_bar = new MenuBar();
    createMenus(m_bar);
  }

  protected abstract void createActions(ActionBar a_bar);
  protected abstract void createMenus(MenuBar m_bar);

  /* ... */
}
```

Resulting Context

- There is no need for additional methods whose purpose is to do the object compositions, which are done through the interface of the exposed objects.
- The way how to use hook methods is intuitive by reading their signature.

Known Uses

The main class of a JHotDraw-based application has to override Composition Hook Methods for plugging *menus* and the *tools* that create the figures. A *viewpart* of an application based on Eclipse RCP has a Composition Hook Method for plugging GUI elements.

Related Patterns

The reader may indeed find a Template Method and this pattern very alike. However, the purpose of a Template Method is more generic, and the hook methods may have purposes other than enabling object composition.

By overriding a Composition Hook Method the variation is achieved through the exposed object. The type of objects that can be composed in the exposed objects is known, and there are methods in the objects' interface specifically for that purpose. Therefore, the body of an overridden Composition Hook Method is predictable in what respects to the

method invocations on the exposed object. For instance, the only purpose of the exposed object of type ActionBar in the given example is to perform register() calls with objects of type Action as arguments. All the implementations of this Composition Hook Method will be similar across framework-based applications.

A Composition Hook Method may become hidden from application developers, so that they will not need to deal with it when building an application. In order to do so, a Self-Pluggable Object is capable of dismissing the need of overriding the Composition Hook Method.

6 Self-Pluggable Object

Context

Framework classes have Composition Hook Methods.

Problem

How to hide a Composition Hook Method from the reuse interface, so that there is one less framework element that application developers have to know about?

Forces

- Usually the type of the objects we want to plug is easy to find. Finding the way how to plug the objects is the most difficult part, since the framework user has to understand the interface of the object where plugging takes place.
- The less hook methods that have to be known and dealt by application developers, the better.
- The lack of documentation may cause application developers to plug objects in wrong locations, resulting in incorrect uses of the framework.

Solution

Develop an abstract aspect that encapsulates the behavior that creates and plugs the object in a Composition Hook Method. Use a Template Pointcut where the fixed part defines the Composition Hook Method and the variable part (hook) is intended to match a subclass of the template class that owns that method. Application developers use a Self-Pluggable Object (Figure 7) by extending the aspect and defining the hook pointcut on the desired context.

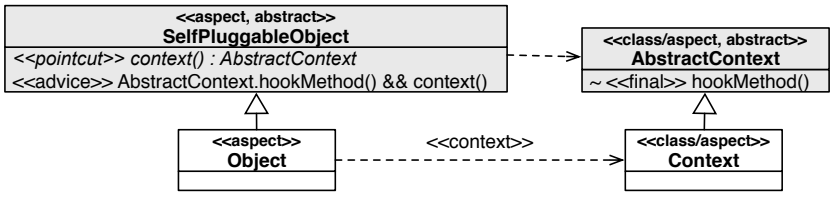


Fig. 7. Self-Pluggable Object pattern.

Example

This pattern is illustrated by presenting a solution for improving the *plugging menus* scenario given in Section 2. The Composition Hook Method is AbstractApplication.createMenus(). Since this method will not need to be overridden by applications, it may have reduced visibility and be locked for overriding, as shown below.

```
public abstract class AbstractApplication {
  /* ... */
  final void createMenus(MenuBar m_bar) {   }
}
```

The following is a reusable aspect for plugging menus. The menu name is given in the constructor, while the menu is created by createMenu() using that name. The hook pointcut is application(). The advice creates the menu and plugs it on the MenuBar object parameter of createMenus(..). Using an independent method for creating the menu facilitates the collaboration with other aspects.

```
public abstract aspect Menu {
   private String name;

   public Menu(String name) {
      this.name = name;
   }

   protected abstract pointcut application();

   after(MenuBar mb) :
      within(AbstractApplication+) && application() &&
      execution(void createMenus(MenuBar)) && args(mb) {
         mb.add(createMenu());
      }

   IMenu createMenu() {
      return new MenuImpl(name);
   }
}
```

The main class of an application (subclass of AbstractApplication) does not need to override createMenus(..). ExampleApplication would be given like shown below.

```
public class ExampleApplication extends AbstractApplication {

}
```

In order to plug a menu, a subaspect of Menu has to be defined. The following is an example of how to plug a menu ("Menu1") in ExampleApplication.

```
public aspect Menu1 extends Menu {
   public Menu1() {
      super("Menu1");
   }

   protected pointcut application() : target(ExampleApplication);
}
```

In case the order of multiple Self-Pluggable Objects of the same type is relevant, precedences have to be used to explicitly declare the order in which the several objects are plugged. The following example shows how it could be declared that Menu1 is to be plugged before MenuX.

```
public aspect MenuOrder {
   declare precedence: MenuX, Menu1;
}
```

The precedence declaration may be given in an independent module as shown, but it can also be given together with the other modules.

Resulting Context

- Application developers no longer have to deal with the Composition Hook Method. Instead, they implement an independent aspect, which defines the hook pointcut.
- Objects can be plugged in other objects incrementally, without the need of modify, inspect, or understand, code related to the object where composition takes place.
- Changing the context where the object is composed can be done only by changing the hook pointcut definition.
- Framework-based applications are adaptable without the need of understanding or modifying source code. Removing a Self-Pluggable Object can simply be done by recompiling the application without its module (e.g. deactivating Menu1 as a compilation unit).

Known Uses

Self-Pluggable Objects in JHotDraw can plug *menus*, *tools*, and *undo* on tools. Self-Pluggable Objects in Eclipse RCP can plug the *toolbar*, *perspectives*, and *viewparts* (an application can have several viewparts, which are organized in different perspectives).

Related Patterns

The Decorator pattern [2] is related to Self-Pluggable Object, in the sense that also allows to add behavior to a class modularly. A Decorator adds behavior dynamically to an object by wrapping it. This implies that when adding a Decorator one has to modify the module that instantiates the wrapped object. A Self-Pluggable Object does not require modifying nor inspecting the module where behavior will be added.

A Self-Pluggable Object can be plugged in another Self-Pluggable Object. This can be done by intercepting the creation of objects in order to plug other objects in them, or by completing Composition Hook Methods, which Self-Pluggable Objects may have. A Self-Pluggable Object may be a Multi-Context Self-Pluggable Object if the object can be plugged in different application contexts. A Self-Pluggable Object may be based on an Abstract Self-Pluggable Object if there are multiple subtypes of the pluggable object. A Self-Pluggable Type Hierarchy merges the type implementations with their abstract composition (i.e. plugging). An Association Object enables the establishment of associations between Self-Pluggable Objects.

7 Multi-Context Self-Pluggable Object

Context

A Self-Pluggable Object is an object that plugs itself in a certain application context. However, there are objects which can be plugged in different application contexts.

Problem

How to develop a Self-Pluggable Object that can be plugged in more than one application context?

Forces

- It is appealing to have everything what is possible to do with an object represented in a single module. By knowing about that module, an application developer knows all that can be done with the object.
- If we would have a Self-Pluggable Object for each application context, there would be multiple modules for addressing a single concept.

Solution

Develop an aspect similar to a Self-Pluggable Object, which has one advice for each Composition Hook Method related with an application context. The hook pointcut is used in the different advices. When using the Multi-Context Self-Pluggable Object (Figure 8), the context is given by the module that is matched by the hook pointcut, implying that only the advice related to that application context will take effect.

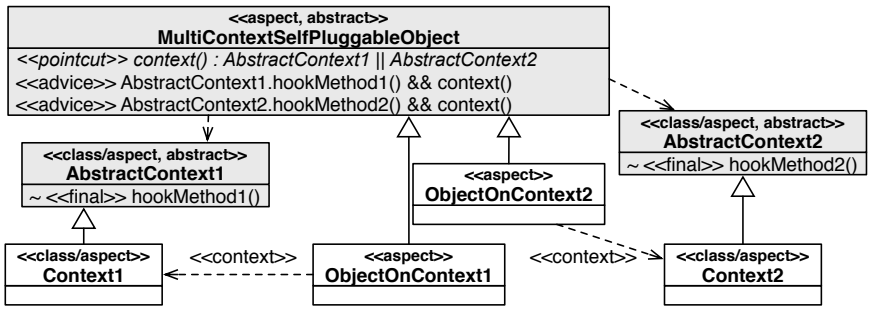


Fig. 8. Multi-Context Self-Pluggable Object pattern.

Example

This pattern is illustrated by evolving the previous example, while addressing the improvement of the *menu context* scenario given in Section 2. Besides the application, a (sub-)menu can also be composed in another menu. Therefore, a menu can be used in more than one application context. The following is a new version of Menu, containing two advices. The first is like in the previous example, while the second is for addressing the composition of menus and sub-menus.

```
public abstract aspect Menu {

   /* to match an extension of either AbstractApplication or Menu */
   protected pointcut context();

   after(MenuBar mb) :
      within(AbstractApplication+) && context() &&
      execution(void createMenus(MenuBar)) && args(mb) {
         mb.add(createMenu());
      }

   after() returning(IMenu m) :
      within(Menu+) && context() &&
      execution(IMenu createMenu()) {
         m.addSubMenu(createMenu());
      }
   /* ... */
}
```

The following module would plug the menu "Menu1" in ExampleApplication (very similar to the example given previously).

```
public aspect Menu1 extends Menu {
   public Menu1() {
      super("Menu1");
   }

   protected pointcut context() : target(ExampleApplication);
}
```

The following module would plug the menu "Menu2" in the "Menu1".

```
public aspect Menu2 extends Menu {
   public Menu2() {
      super("Menu2");
   }

   protected pointcut context() : target(Menu1);
}
```

Resulting Context

- Everything that can be done in an application with an object is achieved through the same module.
- Changing the context where the object is composed, including different context types, can be done just by changing the hook pointcut definition in the object's module.

Known Uses

A Multi-Context Self-Pluggable Object in Eclipse RCP can plug *menus*, whose context may be (a) the application menu bar (conventional

menu), (b) a certain *viewpart* (only shown in there), and (c) a certain *viewer* (appears as a pop-up menu).

Related Patterns

A Multi-Context Self-Pluggable Object is a special kind of Self-Pluggable Object.

8 Abstract Self-Pluggable Object

Context

Objects are plugged using Self-Pluggable Objects. A common case in frameworks is that objects of a certain type (e.g. represented by an interface) may be plugged in an application, and therefore, several subtypes of that type can be plugged in the same way.

Problem

When having a hierarchy of types whose objects can be plugged in an application, if we would have a Self-Pluggable Object for each one, there would exist duplicated code, given that all the objects are plugged in the same way. How to generalize the common behavior that is necessary to plug objects of a certain type?

Forces

- Code reuse should be maximized.
- A Self-Pluggable Type Hierarchy is also suitable to structure Self-Pluggable Objects, but this option is not always viable.

Solution

Develop an aspect similar to a Self-Pluggable Object, but with a Template Advice where the creation of the object to be plugged is done by a Factory Method (abstract method). This Abstract Self-Pluggable Object (Figure 9) should not be visible to applications. Develop one abstract aspect inheriting from it for each type to be plugged, where the implementation of the Factory Method returns the proper object. If application-specific objects of that type are allowed to be plugged, develop also an abstract aspect that implements the type but

which does not implement the methods of the type, so that they can be given in application modules. An application developer may use one of the visible aspects by extending it and defining the hook pointcut. In case the application-specific type is intended to be implemented, the application developer extends the aspect for that purpose, and in addition to the hook pointcut definition, the type's methods have to be implemented.

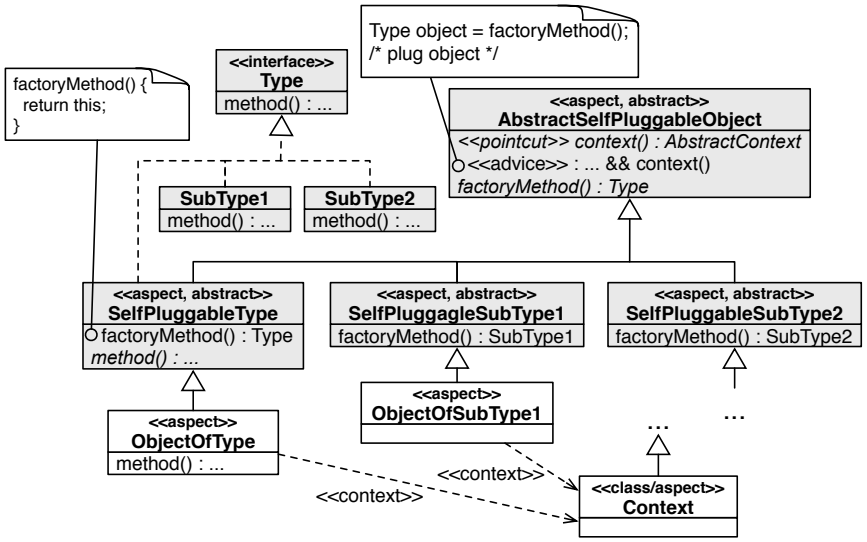


Fig. 9. Abstract Self-Pluggable Object pattern.

Example

This pattern is illustrated with the plugging of *actions* in the example framework. The case of actions is very similar to the *plugging menus* scenario described in Section 2. Applications may include actions by plugging objects of type IAction. The following is an Abstract Self-Pluggable Object for this purpose. Except for the Template Advice, the solution is analogous to a Self-Pluggable Object.

```
abstract aspect AbstractAction {
   protected abstract pointcut application();

   after(ActionBar ab) :
      within(AbstractApplication+) && application() &&
      execution(void createActions(ActionBar)) && args(ab) {
```

```
            ab.register(createAction());
        }

    protected abstract IAction createAction();
}
```

The above aspect can be extended by Self-Pluggable Objects, which can be used by application developers. The following is an example Self-Pluggable Object, based on the Abstract Self-Pluggable Object, for addressing the *exit action*. The aspect overrides createAction() for returning an instance of the framework class ExitAction.

```
public abstract aspect Exit extends AbstractAction {
    protected IAction createAction() {
        return new ExitAction();
    }
}
```

The following module illustrates how Exit could be used, by plugging the exit action in ExampleApplication.

```
public aspect ExitOnExample extends Exit {
    protected pointcut application() : target(ExampleApplication);
}
```

The aspect that allows the plugging of application-specific actions could be implemented like shown below. The aspect implements IAction, but it is up to applications to implement the interface methods (run() in this case).

```
public abstract aspect Action extends AbstractAction
    implements IAction {

    public abstract run();

    protected IAction createAction() {
        return this;
    }
}
```

The following module illustrates how Action could be used. In addition to the hook pointcut definition, the method implementation is given.

```
public aspect ExampleAction extends Action {
    public void run() {
        /* application-specific action implementation */
    }

    protected pointcut application() : target(ExampleApplication);
}
```

Resulting Context

- The plugging of objects of a certain type is generalized. Support for new types can be added simply by developing an aspect module that overrides the Factory Method (e.g. as in Exit).
- The code of the modules that extend the Abstract Self-Pluggable Object still has some redundancy given that the implementation of the Factory Methods across the several modules is very similar (only the name of the class changes).
- The number of subaspects grows along with the number of framework-provided type implementations, implying one more framework module for each one (i.e. the aspect).

Known Uses

In Eclipse RCP, an Abstract Self-Pluggable Object can generalize the plugging of *actions*, which may be either chosen from a set of framework-provided actions or implemented by applications. In JHotDraw there is an analogous case for plugging *commands*.

Related Patterns

An Abstract Self-Pluggable Object serves the purpose of structuring a set of related Self-Pluggable Objects, and consists of an alternative to a Self-Pluggable Type Hierarchy.

9 Self-Pluggable Type Hierarchy

Context

An Abstract Self-Pluggable Object is capable of generalizing the plugging of objects of a certain type, implying that there will exist an aspect for each type. All these subaspects are similar and only differ in the object instance returned by the Factory Method.

Problem

How to avoid the existence of all the similar subaspects, and therefore, to reduce the number of framework modules?

Forces

– In solutions based on an Abstract Self-Pluggable Object, the number of subaspects grows along with the number of framework-provided type implementations. The disadvantage is that the solution implies one Self-Pluggable Object for each pluggable type.

Solution

Merge the implementation of a type hierarchy of default components with the Self-Pluggable Objects that implement the composition of objects of that type. In order to do so, develop an aspect similar to a Self-Pluggable Object that is of the top-most type of the hierarchy (i.e. it declares that it implements that type), while it does not implement the type's methods. Develop a subaspect for each subtype, where the type methods are implemented. These aspects may represent partial type implementations by implementing a subset of the type methods, while leaving the remaining methods to applications. Application developers can use the appropriate member of the Self-Pluggable Type Hierarchy (Figure 10) that implements the type they wish to use. If an application has to include its own implementation of the type, it extends the top-most aspect.

Example

This pattern is illustrated with the same case of the *actions* in the example framework, as it consists of an alternative solution to the one given in Section 8.

The following is a new version of Action that can be used in the same way by application developers (exemplified in Section 8). The aspect is of type IAction, and registers itself as an action.

```
public abstract aspect Action implements IAction {
   protected abstract pointcut application();

   after(ActionBar ab) :
      within(AbstractApplication+) && application() &&
      execution(void createActions(ActionBar)) && args(ab) {
         ab.register(this);
      }

   public abstract void run();
}
```

The following is a new version of Exit that can be used in the same way by application developers (as given in Section 8).

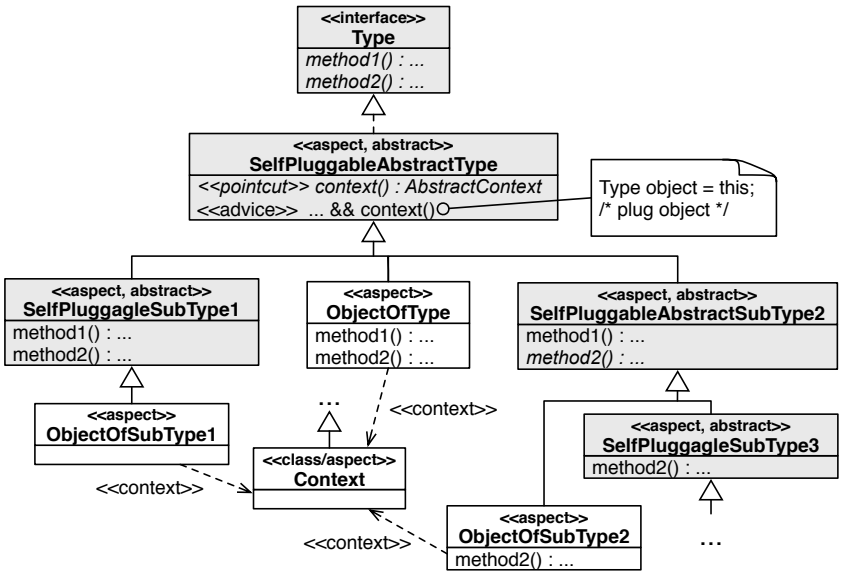


Fig. 10. Self-Pluggable Type Hierarchy pattern.

```
public abstract aspect Exit extends Action {
  public void run() {
    /* the exit action implementation */
  }
}
```

Resulting Context

- Less framework modules when comparing with a solution based on an Abstract Self-Pluggable Object, and more elegant given the nonexistence of all the similar subaspects for each pluggable type.
- The types addressed by the Self-Pluggable Type Hierarchy cannot be instantiated independently.

Known Uses

In JHotDraw, a Self-Pluggable Type Hierarchy is capable of organizing the framework-provided *figures* and *connection figures*. In order to use an application-specific figure, application developers may extend a member of the hierarchy, which may be the top element for implementing

a completely new figure, or a lower one in case if the figure is intended to be based on an existing one.

Related Patterns

If merging the implementation of the types with Self-Pluggable Objects is not possible due to some constraint, a solution based on an Abstract Self-Pluggable Object can be used instead.

10 Association Object

Context

Objects are plugged in an application using Self-Pluggable Objects or Multi-Context Self-Pluggable Objects. The objects plugged in an application may need to have other associations between them.

Problem

The plugged objects are not visible to application developers, so that they can define the associations. How to establish an association between two Self-Pluggable Objects?

Forces

- Having the possibility of managing object associations independently is advantageous because application features relying on associations may be plugged and unplugged without modifying other modules.
- Given that the objects are handled by Self-Pluggable Objects, it makes sense to define associations in terms of these modules.

Solution

Develop an abstract aspect, which when made concrete encapsulates an association between two objects — an Association Object (Figure 11). Use two Template Pointcuts to capture the creation of the objects, each one with its own advice. One advice captures and stores a reference to one of the objects, while the other advice uses that reference to establish the association with its captured object. Application developers can establish an association by defining the hook pointcuts on the modules representing the two objects to be associated. An Association

Object can be defined in terms of an Abstract Self-Pluggable Object or the top-level aspect of a Self-Pluggable Type Hierarchy. This enables that the association can be established between any object whose type is a subtype of the type addressed by either the Abstract Self-Pluggable Object or the Self-Pluggable Type Hierarchy.

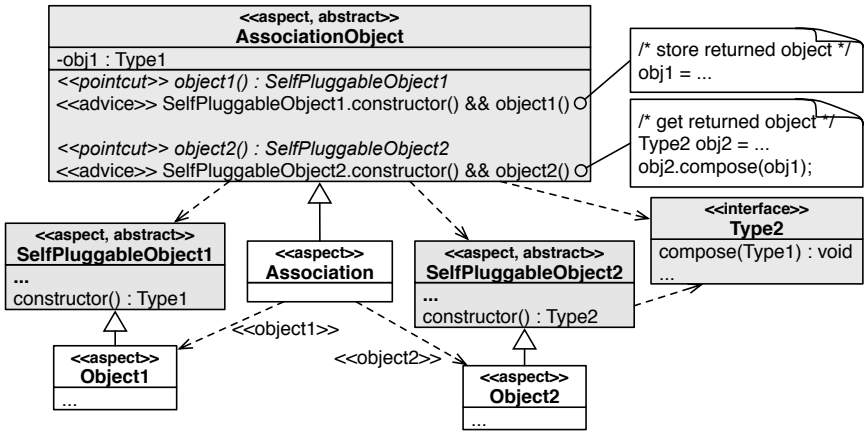


Fig. 11. Association Object pattern.

Example

This pattern is illustrated by presenting a solution for improving the *associating actions* scenario given in Section 2. Below we present the aspect that enables the encapsulation of such associations, assuming the specialization aspect Action from Section 9 and the specialization aspect Menu from Section 7. The first advice captures the instantiation of a subaspect of Action, which is itself an object of type IAction, while the second advice includes the captured action in a menu captured from the execution of createMenu() within a subaspect of Menu.

```
public abstract aspect MenuAction {
   protected abstract pointcut action();
   protected abstract pointcut menu();
   private IAction a;

   after(IAction a) :
      within(Action+) && action() &&
      execution(Action.new(..)) && this(a) {
         this.a = a;
```

```
    }

  after() returning(IMenu m) :
    within(Menu+) && menu() &&
    execution(IMenu createMenu()) {
      m.addAction(a);
    }
}
```

Assuming the Self-Pluggable Objects Menu1 and ExitOnExample given previously, the following aspect implements the association that places the *exit action* on "Menu1".

```
public aspect ExitOnMenu1 extends MenuAction {
  protected pointcut action() : target(ExitOnExample);
  protected pointcut menu() : target(Menu1);
}
```

Resulting Context

- Associations can be encapsulated and managed independently.
- In order to establish an association there is no need to understand the context where the objects are plugged nor any details about their type implementation.

Known Uses

In JHotDraw an Association Object can define the valid source and target *figures* which a *connection figure* may connect. However, the solution is a bit different than the one in the example, since the valid connections are given by overriding a hook method of the connection figure. In Eclipse RCP, Association Objects may link the *actions* and the *toolbar*, the actions and the *menus*, or the *viewparts* and the *perspectives*.

Related Patterns

An Association Object may be adaptable by having Composition Hook Methods, which in turn can be completed by Self-Pluggable Objects.

11 Example Framework Revisited

This section revisits the example framework, taking into account the Modular Hot Spots pattern language given throughout sections 6-10.

Figure 12 depicts the new reuse interface after applying the patterns. We can see the several abstract modules (gray) and their abstract pointcuts. Regarding the *menus*, the example of Section 7 is considered (Multi-Context Self-Pluggable Object), instead of the one given in Section 6. Regarding the *actions*, the examples of Section 9 are considered (Self-Pluggable Type Hierarchy), instead of the ones of Section 8.

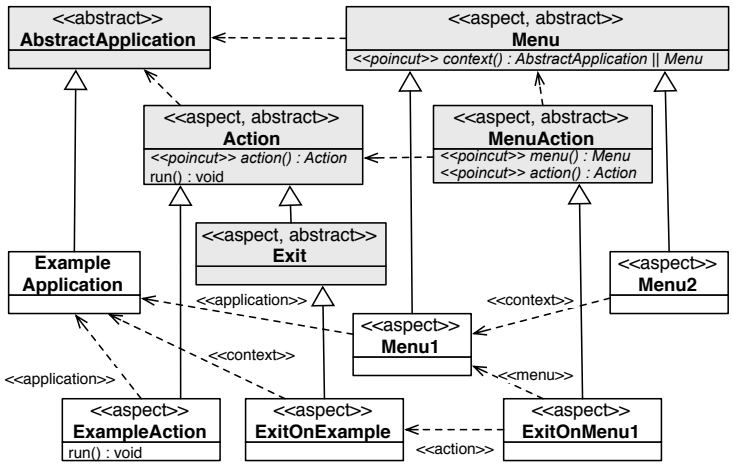


Fig. 12. Modular Hot Spots for the example framework (in gray) and example application (in white).

Figure 12 also depicts the framework-based application based on Modular Hot Spots that was given throughout the patterns. We can see the several application aspects (inheriting from the specialization aspects) and their pointcut definitions. The following topics briefly compare this solution with the conventional one given in Section 2.

1. Each of the application concepts (i.e. *application*, *menu*, *action*, *exit*, and *menu action*) can be used incrementally, where each concept instance is implemented in an independent module. Throughout the pattern examples a framework-based application was given in the modules ExampleApplication (Section 6), Menu1 and Menu2 (Section 7), ExitOnExample and ExampleAction (Section 8), and ExitOnMenu1 (Section 10).
2. Without source code modification, the application may be compiled with subsets of the modules enumerated in (1), obtaining variants of

the application. This issue is particularly important in the context of *software product-lines*. For instance, one could have a variant of the application without the "Menu2", just by not including that module in the compilation.

3. Application features can be removed without understanding source code, as far as one knows which application elements the modules are representing (a fairly basic information that is easy to maintain). This issue facilitates the *maintenance* and *reengineering* of framework-based applications. For instance, suppose that the application implemented by the modules enumerated in (1) needs to be changed for a new version without the ExampleAction. If the task is given to a programmer that was not the one who developed the application in first place, his or her task becomes facilitated, given that only that module has to be identified and removed, while no understanding of the existing code is necessary.
4. The associations between menus and actions can be independently and non-invasively defined.
5. The two hook methods of AbstractApplication, plus the two methods of Menu, of the conventional reuse interface, no longer have to be dealt with by applications. Instead, there are pointcuts that assume their role. The advantages of the latter is that compositions can take place without modifications and inspection of the target modules.
6. The two classes MenuBar and ActionBar are no longer relevant for the application developer.

The items (1), (2), and (3) are related with the improvement of the *plugging menus* and *menu context* scenarios given in Section 2. Item (4) is related with the improvement of the *associating actions* scenario also given in Section 2.

12 Conclusion

This paper presented Modular Hot Spots, a pattern language for helping on the task of developing framework reuse interfaces with a higher abstraction level concerning the development of framework-based applications. The application of the given patterns relies on aspect-oriented programming primitives. However, the required knowledge of this programming paradigm is small, if we consider the whole set of primitives that these languages offer.

Modular Hot Spots can form a black-box reuse interface with a higher level of abstraction than conventional black-box reuse interfaces.

Black-box frameworks are pointed out as adequate for having an accompanying Visual Builder [7] for generating framework-based applications from high-level domain-specific descriptions. We argue that such Visual Builders can be developed more easily if Modular Hot Spots are adopted, given that the code of the applications is able to resemble more closely the concepts and relationships of a given domain.

Acknowledgements

We would like to thank our EuroPLOP'08 shepherd Uirá Kulesza, and the Writer's Workshop participants Paul G. Austrem, Dietmar Schütz, Diethelm Bienhaus, and Jürgen Salecker, for the valuable suggestions for improving this paper.

References

1. Eclipse Foundation. AspectJ programming language. http://www.eclipse.org/aspectj, 2007.
2. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., 1995.
3. S. Hanenberg, A. Schmidmeier, and R. Unland. Aspectj idioms for aspect-oriented software construction. In *8th European Conference on Pattern Languages of Programs (EuroPLoP)*, 2003.
4. R. E. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1:22–35, 1988.
5. G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings European Conference on Object-Oriented Programming*, 1997.
6. J. McAffer and J.-M. Lemieux. *Eclipse Rich Client Platform: Designing, Coding, and Packaging Java(TM) Applications*. Addison-Wesley Professional, 2005.
7. D. Roberts and R. E. Johnson. Evolving frameworks: A pattern language for developing object-oriented frameworks. In *Pattern Languages of Program Design 3*. Addison Wesley, 1997.
8. A. L. Santos, A. Lopes, and K. Koskimies. Framework specialization aspects. In *AOSD '07: Proceedings of the 6th International Conference on Aspect-Oriented Software Development*, 2007.
9. SourceForge. JHotDraw framework. http://www.jhotdraw.org, 2006.
10. O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: An aspect-oriented extension to C++. In *Proceeding of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, 2002.
11. A. Weinand, E. Gamma, and R. Marty. Design and implementation of ET++, a seamless object-oriented application framework. *Structured Programming*, 1989.

Patterns for Managing Data in Complex Automatic Identification and Data Capturing Environments

Diethelm Bienhaus
Institute of Nanostructure Technologies and Analytics / Technological Electronics
University of Kassel, D-34132 Kassel, Germany
and
Department of Systems Engineering
University of Cooperative Education Nordhessen, D-35066 Frankenberg, Germany
bienhaus@uni-kassel.de, d.bienhaus@ba-nordhessen.de

Keywords: Automatic Identification and Data Capture (AIDC), Automatic Identification (AutoID), Radio Frequency Identification (RFID), Internet of Things, Unique Identifiers, Data-On-Tag, Data-On-Network, Middleware, Edgeware

1. Introduction

Nowadays production and distribution processes are controlled to a large extent by information processes. Interconnecting the flow of material and information promises to increase efficiency and quality of business activities while reducing costs. Production planing and control systems and enterprise resource planning systems can optimize processes within companies.

Supply chain management comprises the whole process of planning, implementing, and controlling the operations within a supply chain. The overall goal is to satisfy customer requirements as efficiently as possible while saving time and resources on the other hand. Controlling all logistic processes from raw materials suppliers to manufacturers and retailers to the consumer needs transparency of the process steps. Managing the whole product life cycle from point-of-origin to the point-of-consumption and disposal or recycling needs accurate information about products in each process state and at the appropriate location.

Automatic Identification and Data Capture (AIDC) or Automatic Identification (AutoID) comprise techniques to automatically identify objects, to collect associated product data, and to propagate that data to back-end software applications like Enterprise Resource Planning systems. Typical automatic identification technologies are bar codes, Radio Frequency Identification (RFID) and smart cards. These techniques identify assigned properties while biometrics, optical character and voice recognition utilize natural properties.

RFID has its origins more than fifty years ago. But due to recent developments this technique is now available with higher quality and more functionality at lower costs. Especially in logistics a large-scale commercial application is expected. In [Hen09] an worldwide market increase of 25% p.a. is expected.

RFID technology comprises several aspects: infrastructure and architecture, hardware like tags or transponders and readers, integration on the physical layer up to IT system integration: SCM, ERP, MES and warehouse management systems.

System architects, technology integrators, process designers and engineers in charge of implementation and system integration face several challenges. On the physical level first of all "material things" have to be identified. Then the captured data can be processed, filtered and forwarded to applications which perform planning and controlling tasks. Questions concerning appropriate data formats, efficient data transfer, data integration, lookup mechanisms, and others arise in AutoID environments. This pattern collection introduces patterns dealing with aspects of those problems.

This collection is a continuation and extending of a work starting with "A Pattern Language for Process Optimization with Smart Object Identification" at EuroPLoP 2005 [Bie05] and "Patterns for Unique Product Identification" [Bie08].

2. Overview

Figure 1 illustrates the relationships among the patterns presented in this collection. Starting point of this collection is the need for centralized product data management as a basis for planning and controlling systems. The pattern Identifiers Point To Data introduces a solution to that problem. Applying the pattern may result in situations where new challenges arises: capturing identification data of each individual product at many process steps and at several times bears the risk of producing a large amount of data at low information level. A solution for that problem is described in the pattern Business Events.

A decentralized data storage has benefits in applications where access to a network infrastructure is temporarily or in principle not available. In such scenarios Data Accompanying Products is an appropriate solution. Storing product data decentralized attached to the product and at the same time centralized on servers can easily result in data inconsistencies. Synchronised Data Location explains how to cope with that problem.

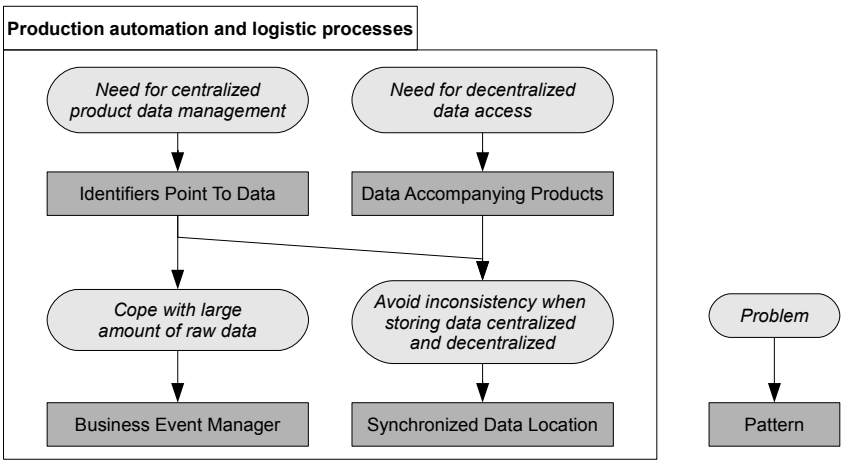


Figure 1 – Overview Graph

Identifiers Point To Data

Context Identifiers Point To Data is necessary to enable seamless access to product data within a single company's internal production processes as well as in case of supply chains where several companies are involved.

Problem ***How to give access to detailed product data in distributed production and logistic networks while only an identifier is attached to the product?***

Forces

- In production and distribution processes several partners needs to gain access to detailed product data. If the product data representation is attached to the product itself the describing information can only be obtained when the product is present.
- Product data not only has to be accessible but also has to be updated and maintained during the whole product life cycle from raw material suppliers to customers and finally to disposal or recycling. Optical data representations are printed once and afterwards unchangeable. Modifications can only be done by replacing an old label or product accompanying description by a newer one.
- Machine readable data has to be stored in such a way that it is assigned to the describing product. Typically a label is fixed on a product or it's packaging. Alternatively the product itself is marked (Direct Part Marking) e.g. by means of laser engraving. Relatively cheap RFID transponders store about a hundred Bits (EPC specifies a 96 Bit encoding) up to one or two kBits. The amount of data is restricted.
- An unique identifier allows distinction of product types or even individuals. But this is not sufficient if a detailed product description is necessary e.g. for customers or due to documentation reasons.
- Increasing complexity of several companies spanning production processes and deep supply chains raise the demand for product data accessible beyond company internal and isolated IT systems like Production Planing Systems (PPS) and Enterprise Resource Planning (ERP) Systems.

Solution ***Establish a network-based infrastructure that consists of components providing services for registering resources with an associated data lookup service which enables storage and query of data along with their identifiers (keys). The services may be be folded into one single piece of software or can be distributed over the Internet.***

In real world physical objects populate the environment. Products or transport units are a specific type of physical objects as results of planed and controlled production processes or as part of such processes or as elements of transportation and storage.

Information technology supports controlling of technical or administration processes. Processing and storing data about the real world needs abstract model of reality. Objects of the real world are abstracted in the information systems through the use of symbols (names, labels).

Abstraction is necessary because the real world cannot be represented in each detail, but relevant sections of reality have to be chosen. Data describing properties of the physical objects may be extensive. Often several parts of organisations or individuals especially customers and users are interested in the data belonging to a product or transport unit, even if the physical object itself is not present.

Rationale Accessibility to product data can be gained by saving and processing the data in decentralized databases. If these databases can be accessed over the Internet most of the involved partners will find the data easily. An identifier has to be assigned to products or transport units and to be carried along with them which works as a "pointer" to it's belonging data. In this sense the identifier externalises the identifier that is necessary internally in databases to mark each entry uniquely. RFID is seen as an enabler so that things of the real world have a representation in the "virtual" world of the Internet [KBM+00], [Bie05].

Consequences

Benefits

- Product data can be accessed independently of the products presence.
- Only the identification number has to be represented or stored. Hence printed labels or inexpensive passive RFID tags are sufficient.
- Generation, representation and assignment of identifiers for products or transport units is well standardized. The standards specified for reading devices, middleware, and data manipulation grant compatibility and ensure a long-term use of AIDC technologies like barcode, two-dimensional symbologies and in an increasingly regarding RFID.

Liabilities

- Data accessibility is highly dependent on the availability of network infrastructure.
- Centralized data storage and processing has to be reachable from every decision point within the production or logistic processes. Failure of the centralized IT system or network components result in a stop of the process. Redundancy can reduce the risk of system standstills, but raises costs.
- Network traffic increases with the number of items which are stored and have to been searchable. The effort of resolving identifiers, searching the data and delivering it to the requester grows proportional to the amount of item records.
- Centralized data processing induces demands for data access regulations. Security issues may be raised by producers and customers as well: To avoid business espionage producers need to hide product details from their competitors. On the other hand customers have concerns that their consumer's behaviour may be recorded and then misused by unauthorized parties.

Known Uses

A well known use are cash registers at checkout points in supermarkets which scan the bar code on products to obtain data like price or weight. The data can be stored decentralized at the cash register itself or is stored on a centralized system for the reason of easier updates.

In logistics nearly each returnable transport item like pallets or containers are assigned a global shipping item number such that the transport units can be tracked. Interoperability and ease of use need standardisation. Global Standard 1 (GS1) has defined a scheme for globally applicable identifiers for trade items (products and services), the Global Trade Item Number GTIN. A whole family of data structure are specified for different application areas. In case of consumer products the EAN-13 bar code is very popular especially in Europe [GS1]. The symbol encodes 13 numbers and is divided into four parts:

- System code: two or three digits representing the country where the manufacturer is registered. To cover the International Standard Book Number ISBN and the International Standard Serial Number ISSN the starting three numerals 978 resp. 979 are used.
- Manufacturer code: four or five digits such that system code and manufacturer code are 7 numerals in sum.
- Product code: five digits given by the manufacturer (normally the serial number).
- Check digit: one digit for the check sum.

Figure 2 gives an example of a EAN bar code (code generator available on the Internet: http://www.barcoderobot.com)

Figure 2 – Example of an EAN-13 Bar Code

The EAN identifier can be scanned to look up product information especially a description and the product price in a database. As a service for consumers GS1 provides a web site on the Internet (`http://directory.gs1.org/gtin/search`) where the product code owner and item information can be searched for.

Figure 3 shows the result of looking up the EAN number 4006381105262 which the author found on a pen. (The Look-up tool is available on the Internet: `http://directory.gs1.org/gtin/search`)

Figure 3 – EAN-13 Item Number Resolved

Several courier-, express- and parcel (CEP) service provider offer customers freight tracking services. In each relevant step in the logistic chain from sender to handler and finally the the recipient is documented by capturing identification data and associating it to the fulfilled transaction. As a service customer can look up information about their shipments.

Data Accompanying Products

Also Known As: Data-on-Tag

Context Application of Data Accompanying Products enables process automation where data about physical objects like products or pallets is necessary.

Problem ***How to provide machines access to product data without presence of a network infrastructure?***

Forces

- A centralized data storage allows data access from several distributed decision points in production and logistic processes. If the necessary network is not available or if the centralized data storage and processing system fails the whole process stops.
- Identifiers attached to physical objects in the real world operate as pointers to resource in the "virtual " world. If the only information carried along with a physical object is the identifier then no detailed data can be obtained between points with no network access.
  But in case of unplanned events more detailed data like product contents may be necessary which cannot be obtained abroad from the normal process paths.
- Network traffic and lookup times increase with the amount of items for which detailed data is requested.

Solution ***Use a machine readable representation or storage technique carried along with the product itself such that product related data accompanies the physical object.***

In order to attach data to a physical object in a machine readable manner an appropriate representation is necessary which is capable to store data and allows at least a reading access.

Read only techniques can be distinguished due to the used storage media and communication method. The most common solutions are:

- Optical representation with transfer in the frequency range of visual light. Data is encoded as linear or two dimensional arranged geometrical patterns like rectangles, squares or circles. The encoding is based on the shape of the geometrical element and/or it's location.
  Typically the patterns are printed on adhesive labels which are then attached to the physical object. Alternatively a direct part marking by means of Laser engraving or other surface processing is applied to prevent removing of the data.

- Data storage in digital memories and communication devices on chips. Data transfer is performed via radio frequencies. On principle, storage capacities of digital memories can be very large but data transfer then is time consuming. While printed or directly marked symbologies are immutable except for replacing the label or modifying the marked surface, digital memories provide the opportunity of not only reading from but also writing to the chip.

Rationale Product data accompanying the the product itself grant access to information without a network infrastructure and a centralized data processing [DMS07].

Attaching human readable information to products is well established for decades. Examples are product sheets or instructions for use which are printed separately and packaged together with the product. Shorter instructions and descriptions are printed on labels and fixed on the product. Textual representations are not obsolete and to be replaced by machine readable data. In fact machine readable information augments the "audience" such that devices can get read the data which is then can be exploit to automatically control process steps or is displayed to humans. Attachment of data beyond simple identifiers raise the demand for higher storage or representation capacities. In case of optical identification techniques data capacity was stepwise enhanced in the evolution from linear codes – bar codes – to stacked codes and then to two-dimensional symbologies. A widely used two-dimensional symbology is the Data Matrix code which can contain up to 3116 numeric or 2335 alpha-numeric characters.

The digital memory integrated in RFID transponders feature a capacity of typically about 100 Bit sufficient for identifiers up to several kBit. It is guessed that Moore's Law just begins regarding RFID transponders.

Consequences

Benefits

- Access to product data can be accessed independently of the products presence.
- In case of re-writeable data storage labels like RFID transponders relevant data can be collected on the path a product is undergoing.
- Object related data allows managing of processes without a centralized controlling system.
- Data is available without an IT infrastructure and without the need for a centralized storage and processing system.

Liabilities

- Detailed data is only accessible when the described physical object is present.
- Data storage capacities limit the amount of information. The level of details hence is restricted and not all relevant data might be kept.

Known Uses

Automation of Production Processes

Siemens Amberg produces transformers which can be individually customized. Figure 4 shows a transport unit used in the transformer production process at Siemens Amberg. Individual configuration data and necessary production steps are stored in the transponder at the beginning of the production line. After that the transport unit is directed through the production and the specified steps are performed automatically without a centralized production execution system.

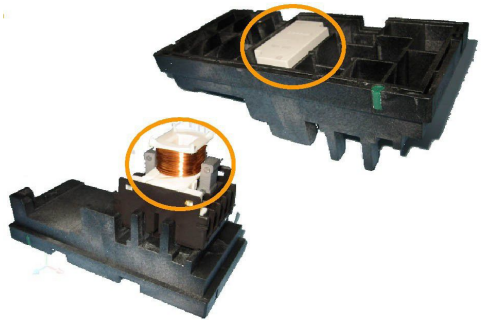


Figure 4 – Transponder storing configuration and production related data attached to transport unit. (Source: Siemens)

Further applications of RFID in process automation are Ford's manufacturing process at the Essex Engine Plant in Windsor, Ontario, and the aircraft maintenance at Boeing and Airbus [ZGYP03], [ZGP03].

Tracking and Tracing of Pharmaceutical Products

In the United States the Food and Drug Administration (FDA) forces the introduction of a so called *ePedigree* for pharmaceutical products. In California by law prescription drugs have to be accompanied by an electronic pedigree.

In the US RFID is the preferred tracking technology. Due to problems of wide spread application of RFID transponders, hybrid pedigree systems incorporating paper and automatic identification technologies like RFID or optical codes are in use as shown in figure 5.

Figure 5 – RFID label attached to a pharmaceutical package

Synchronized Data Location

Context Synchronized Data Location is needed if data is alterable at both the data carrier attached to the product and in centralized storage.

Problem ***How to avoid data inconsistencies if product data accompanying a physical object itself and the mirrored data on centralized servers are changed independently?***

Forces

- A centralized data storage allows data access from several distributed decision points in production and logistic processes. But if the infrastructure is not available or if the centralized data storage and processing system fails the whole process stops.
- Carrying product data along with the product itself provides access to that information independently from a network infrastructure – stand-alone reader devices are sufficient. But due to the decentralized storage technique necessary information is not permanently and accurately available for tasks on a system wide level like production planning and controlling.
- Integration of all reader devices in a network infrastructure can help to forward product related data to centralized processing services. In this case data is pushed forward each time a physical reading process happens. On the level of business-logic applications the need for information about the status of physical components is triggered by software execution. Hence, data generation and information processing are not synchronized. But the ability to react immediately to physical events needs accurate process data.
- Storing product data in both ways – attached to the product itself and on centralized servers – often results in data inconsistency. E.g. updating the product data attached to the product at each process steps provides detailed information about a product's history at any time and anywhere just having a mobile reader device at hand. But that information tends to differ increasingly from the centralized stored data.

Solution ***Synchronize product data carried along with the product and the centralized stored data by means of synchronization rules.***

Such rules define actors and execution details of data interchange processes. Important actors are data producer and consumers who are assigned updating and writing access rights. Synchronisation with the centralized system is mostly based on data versioning and event-based rules.

Rationale

In automation and logistics planning and controlling applications need data related to products and process progress. Modern AutoID technologies support those applications. The overall aim is efficiency based process transparency: to know as much as needed about state and location of products or transport units. Traditional identification data like Barcode labels or imprints are unchangeable. Hence no data inconsistency can occur (expect in case of damage or replacement of the ID). Radio-frequency identification allows storage of data far beyond simple identification numbers. Data Accompanying Products explains benefits of decentralized data storage. The resulting risk of inconsistent contents between separately located data not only arises in the case of modern automated identification and data capturing technologies: it is a general problem that occurs in backup scenarios and distributed systems. Hence AutoID infrastructures can benefit from solutions established in other IT application areas.

Consequences

Benefits

- Centralized stored data can more accurate represent the actual state of physical things within the monitored process.
- Decision and controlling systems can react more promptly to occurring changes within processes.
- In case of writeable storages attached to products like writeable RFID transponders central data updates can be downloaded and deployed to the de-centralized memories on-site.

Liabilities

- Data consistency is determined by the completeness of the synchronization rules. Unaccounted exceptions may still result in data inconsistencies.
- If the necessary network infrastructure is not available no synchronisation can be performed. Data accuracy and hence quality on the centralized storage is dependent on refresh periods.

Known Uses

Based on [Sch02] researchers at University of Bremen / BIBA developed the concept of “Data Contracts” for AutoID applications. Data Contract specify how to update and match data between locally and centralized stored data during a product's life cycle.

Updating and synchronisation of product data during its whole life cycle is the aim of Product Lifecycle Management PLM. From production to distribution, reselling, consumption and recycling all generated and necessary data about the products of a company have to be managed consistently and have to be provided to controlling and planning systems [AG06]. Automatic identification techniques contribute to PLM on the narrow layer of plant automation and production or distribution logistics.

Business Event Manager

Context Business Event Managers bridge the gap between streams of raw sensor data and applications on the business-logic level.

Problem ***How to perform process controlling efficiently while on-site reading devices can only deliver raw sensor data which might even be redundant?***

Forces

- Automatic identification techniques sense physical properties of things like given properties or attached identifiers. Controlling of production and logistic processes is based on logical rules which determine control flows.
- Auto-ID technologies will enable businesses to move from linear and manual supply chain planning and execution to an event-driven, adaptive supply network. But devices like bar code scanner and RFID readers can generate multiple readings of the same physical object.
- The occurrence of a reading event is not necessarily a business relevant event.
- Propagation of each single reading event to the planning system may result in high network traffic.

Solution ***Augment the automatic identification and data capturing infrastructure by additional on-site software that performs a preprocessing of the raw data and propagates business relevant events to the process controlling applications. Such software components act as business event managers.***

Business event managers have to react on real-time events and information, to propagate alerts, and to provide services for essential reading and - if applicable – writing of information to other information systems. As shop-floor near components they have to provide interfaces for reader devices like scanners or RFID readers and other devices like sensors. To be easily integrable those components need standardised network interfaces. To be accessible from the business-logic level business event managers have to support standard communication protocols like Simple Object Access Protocol (SOAP).

Possible field of applications raise with abilities of general-purpose event routing, collating, and filtering.

Rationale Business event extractors and managers hide hardware specific interfaces and extensive streams of raw data. Furthermore, they can accomplish hardware configuration tasks and maintenance related services.

From the physical layer consisting of hardware devices like scanners and readers only physical signals can be obtained. Devices on this layer have to been managed e.g. by providing appropriate software like drivers.

Signals have to be transformed into data due to specific protocols and grammars in order to be transferred to the next layer. So at least two tasks have to be done on the hardware layer: device management and data extraction and communication [GH06].

The top layer of business process management can be seen as the opposite of the hardware related layer on the bottom. Back-end systems like Enterprise Resource Planning (EAI) solutions, Manufacturing Execution Systems (MES) or Supply Chain Management (SCM) systems need to know the status of the underlying processes from a business perspective. Hence business process relevant events or alerts are necessary for decision making and then commands are sent down to the active process elements to control progress. So extraction of business relevant events and command propagation are the tasks to link the business controlling layer and the layer of process near sensors like readers and actors.

Consequences

Benefits Assignment of specific tasks and functionality to specialized components supports flexibility and re-usability. Especially in case of multi-tier architectures the necessary processing power is at the point where the relevant data has to be captured and preprocessed.

Liabilities The increasing amount of components and communication overhead are time costly and decrease performance. Since more processing power is allocated at distributed hardware components the installation costs of the whole infrastructure grow.

Known Uses

In case of complex production processes or supply chains several distributed and heterogeneous components have to be integrated. This is the task of middleware which is referred to as Enterprise Application Integration (EAI). In between of those layers several functionality is necessary:

- Preprocessing of the raw data received from devices,
- data filtering and aggregation,
- context-based event extraction,
- propagation of business relevant events and
- connectivity components for communication.

This intermediate layer, which might be subdivided, is known as “edgeware”. On the edgeware layer several tasks have to be done. Devices are distributed at decision relevant points of the process. Often inhomogeneous hardware components are in use. Software components specialized in those different tasks and adapted for the different hardware devices are the constituent parts of the edgeware layer.

Edgeware software is a good example where to apply the Layers [BMR+96] and Pipes and Filters [BMR+96] architectural patterns.

Distinct parts work different levels and specific protocols can be used to access the services contained at each layer.

In case of a single tier architectural approach all software functionality for data and device management is integrated as a single software component with a layered internal structure. Figure 6 illustrates a single-tier edgeware architecture.

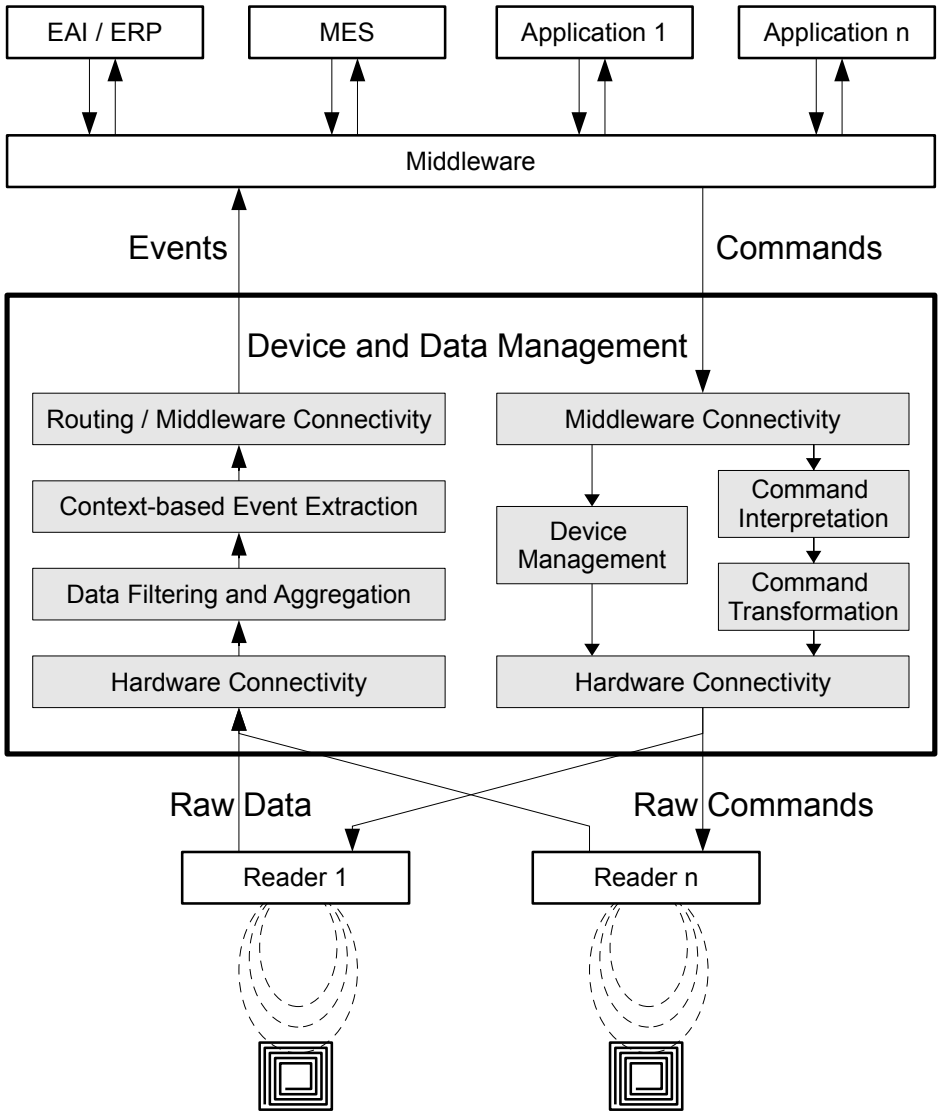


Figure 6 – Single-tier Edgeware Architecture

In production plant automation and in logistic applications reading devices are distributed. In such a setting a multi-tier architecture is more appropriate, as shown in figure 7. In this architecture an additional service bus separates command interpretation and event extraction from hardware specific functionality. The latter consists of command transformation into device specific formats as well as data filtering and aggregation. In [Lea04] several industrial applications of one-tier and multi-tier architectures are introduced.

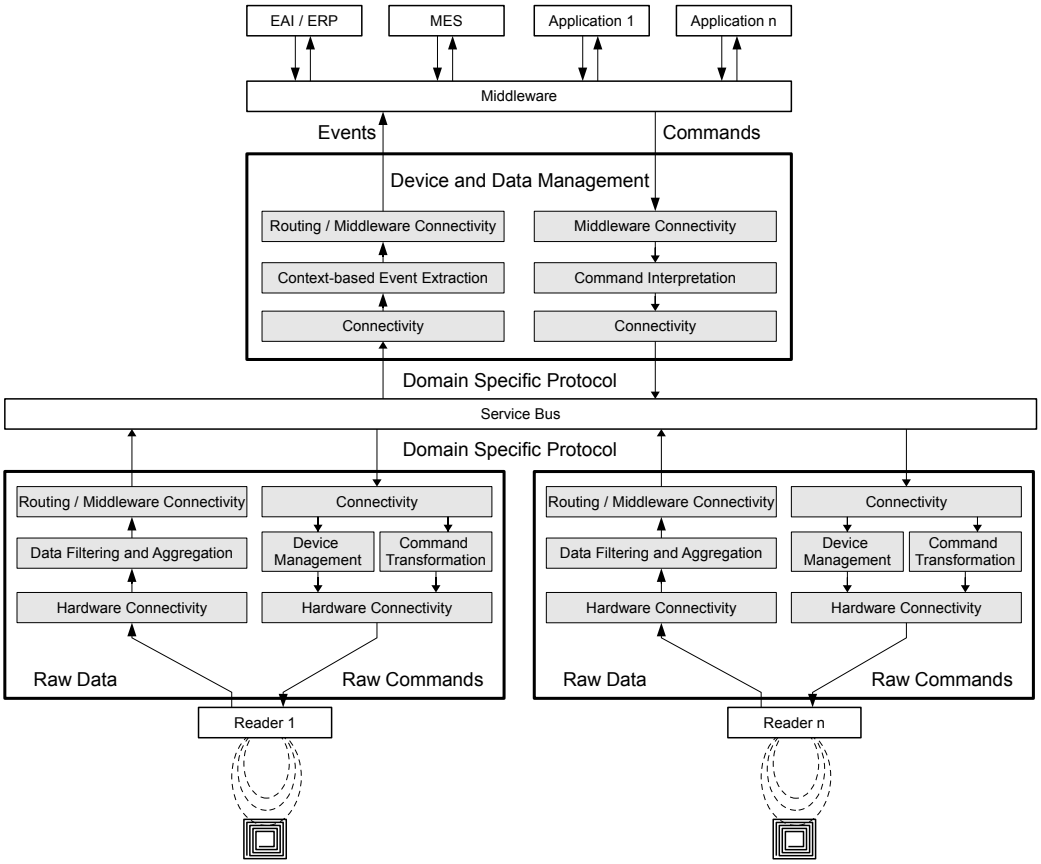


Figure 7 – Multi-tier Edgeware Architecture

GS1 aims on a standardisation of integration architectures. That standardisation defines the *The Savant Architecture* [LNE05], [CTAO03] where interface components on the hardware near layer perform simple data processing in order to extract *Application Level Events* from raw sensor data.

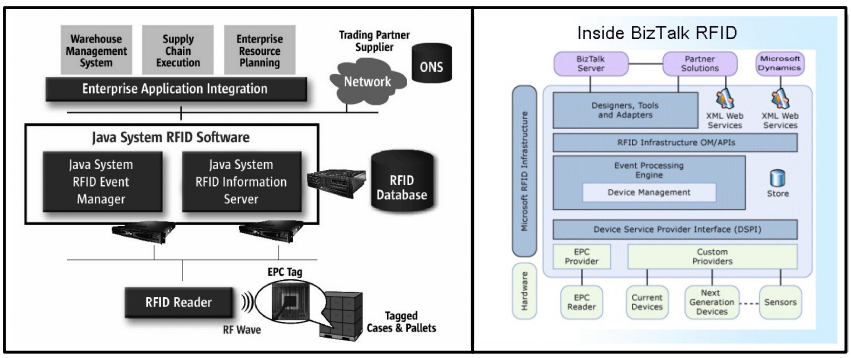


Figure 8 – Event Manager Software Components in RFID Middleware
Left: Sun's RFID Middleware (Source: Sun Microsystems)
Right: Microsoft's BizTalk Architecture (Source: Microsoft)

The architectural approach of extracting business events is implemented in middleware platforms like Microsoft BizTalk [Sch06] or Sun's RFID middleware [Sun05]. In both architectures specialized software components ("Event Managers") gather information from the RFID Readers, filter the information, and provide information as messages to systems on the business layer like ERP systems. Figure 8 illustrate how the Event Manager and Information Server fit into a network-based AutoID architecture.

3. Acknowledements

I'd like to thank my EuroPLoP 2008 shepherd Christian Kohls for his helpful feedback and inspiring suggestions.

References

[AG06] Abramovici, M. and M. Ghoffrani: *PLM - ein Thema auch für KMUs*. Digital Engineering, 2006.

[Bie05] Bienhaus, Diethelm: *A Pattern Language for the Network of Things*. In *Proceedings of 10th European Conference on Pattern Languages of Programs (EuroPlop 2005)*, Irsee, Germany, 2005.

[Bie08] Bienhaus, Diethelm: *Patterns for Unique Product Identification*. In *Proceedings of 12th European Conference on Pattern Languages of Programs (EuroPlop 2007)*, (to be published) 2008.

[BMR+96] Buschmann, F., R. Meunier, H. Rohnert, P. Sommerlad and M. Stal: *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley, New York, 1996.

[CTAO03] Clark, Sean, Ken Traub, Dipan Anarkat and Ted Osinski: *Auto-ID Savant Specification 1.0*, 2003. `http://www.nepc.gs1.org.sg/epcglobal/stdsdocs/WD_savant_1-0_20030911.doc` (17.02.2009).

[DMS07] Diekmann, Thomas, Adam Melski and Matthias Schumann: *Data-on-Network vs. Data-on-Tag: Managing Data in Complex RFID Environments*. In *HICSS '07: Proceedings of the 40th Annual Hawaii International Conference on System Sciences*, page 224a, Washington, DC, USA, 2007. IEEE Computer Society.

[GH06] Gillert, Frank and Wolf-Rüdiger Hansen: *RFID für die Optimierung von Geschäaftsprozessen: Prozess-Strukturen, IT-Architekturen, RFID-Infrastruktur*. Hanser, München, 2006.

[GS1] GS1: *EAN Specification*. Available on the internet: www.gs1.org.

[Hen09] Heng, Stefan: *Heng, Stefan,RFID Chips: Enabling the Efficient Exchange of Information(February 6, 2009). Deutsche Bank Research Paper. Available at SSRN: http://ssrn.com/abstract=1339543*. Deutsche Bank Research Paper, February 6 2009. Available at SSRN `http://ssrn.com/abstract=1339543`.

[KBM+00] Kindberg, Tim, John Barton, Jeff Morgan, Gene Becker, Debbie Caswell, Philippe Debaty, Gita Gopal, Marcos Frid, Venky Krishnan, Howard Morris, John Schettino and Bill Serra: *People, Places, Things: Web Presence for the Real World*. WMCSA 2000, Monterey, USA, December 2000.

[Lea04] Leaver, Sharyn: *Evaluating RFID Middleware*. Technical Report, Forrester Research, Inc., 2004. Available on the Internet: http://www.bauer.uh.edu/rfid/ForresterRFIDwave.pdf (18.02.2008).

[LNE05] Leong, Kin Seong, Mun Leng Ng and Daniel W. Engels: *EPC Network Architecture*. Technical Report, Auto-ID Labs, Massachusetts Institute of Technology and Adelaide, 2005.

[Sch02] Schichtel, M.: *Produktdatenmodellierung in der Praxis*. Carl Hanser Verlag, München et al., 2002.

[Sch06] Schwartz, Karen D.: *BizTalk RFID: Making RFID Deployments Easy, Simple and Economical*, June 2006. `http://msdn.microsoft.com/en-us/library/aa479354.aspx` (17.02.2009).

[Sun05] Sun Microsystems: *The Sun Java System RFID Software Architecture - A Technical White Paper*, March 2005. `http://www.sun.com/solutions/documents/white-papers/re_EPCNetArch_wp_dd.pdf` (17.02.2009).

[ZGP03] Zhekun, Li, Rajit Gadh and B.S. Prabhu: *A Study of RFID Smart Parts*. Technical Report, University of California, Los Angeles, Wireless Internet for the Mobile Enterprise Consortium, 2003.

[ZGYP03] Zhekun, Li, Rajit Gadh, Fan Yujin and B.S. Prabhu: *Study of potential of Wireless Internet Technologies in Manufacturing*. In *The Third International Conference on Electronic Commerce Engineering (ICeCE2003)*, 2003.

INTELLIGENT SUBJECT – adapting OBSERVER with push model and filters to handle divergent update needs

Paul G. Austrem
Dept. of Information Science and Media Studies
University of Bergen, Norway
paul.austrem@infomedia.uib.no

Abstract

The OBSERVER design pattern is one of the most widely used patterns from the original GoF book [1]. With the proliferation of mobile devices in worklife and information systems serving data to such devices is paramount to maintaining data integrity in a work process. The idiosyncracies of mobile devices have placed new requirements on the mechanisms for updating resource limited clients[1] with an OBSERVER style solution. This work provides an adapted pattern named INTELLIGENT SUBJECT that allows for a SUBJECT side filtering mechanism to avoid propagating all updates to all OBSERVERS if the cost of notification is high. This cost could be either due to resource or network constraints. OBSERVERS define threshold values, and are only notified when the data value is changed beyond their individual threshold. The pattern introduces slightly more complexity, but allows for a separation of concerns on the SUBJECT side and a life of blissful ignorance on the OBSERVER side.

Introduction

Currently mobile devices are being increasingly used as integral parts of day-to-day operations in many business areas, being employed by healthcare workers [2], ticket takers on trains, as well as suggested uses for construction workers [3]. This is complimented by the increased development of mobile devices supporting constant network connectivity (through technologies such as Wi-Fi and/or HSDPA) for broadband data transfer speeds, along with GPS technology for location data[4] . These technologies pave the way for *mobile knowledge workers* to utilize information on-the-go to improve their workday, as they can now access a centralized information system or data source through their network connectivity, and enrich information retrieval techniques with contextual information through the use of GPS location data. This work uses the definition of a mobile knowledge worker as a person who does not have a stationary workplace and who is dependant on updated information in order to perform their work tasks. Note however, that the domain of mobile knowledge workers is not normative for the pattern, it is exemplary. The pattern may of course be used in other situations and contexts, the specific domain is applied here because it brings forth many of the benefits of this adaptation of the original OBSERVER pattern.

How data is used by different applications on a mobile device may vary. For instance maybe you are running several different applications simultaneously on your mobile device. On a mobile device (hereafter referred to as a `Client`) with constrained resources, there should be an aim to minimize unneccessary inter-process calls such as with applications actively polling a shared resource on the `Client`[5, 6].

Both [5] and [6] offer an alternative to making the `Client` responsible for retrieving location data in a data pull-manner. In this paper, the OBSERVER pattern [1] is used to allow `Client` applications to register with, for example, a `LocationManager` and receive either periodic

[1] Devices that are battery powered, have limited memory or have reduced processing power.

updates [5], or updates whenever the user has moved beyond a set proximity [6]. The pattern has also been applied in the Symbian OS for mobile devices as part of the MVC pattern. It is in this context used to notify views of updates/changes to the model.

The OBSERVER pattern is used to offer this functionality. The pattern is one of the most widely applied patterns in software today. It allows a system to achieve consistency among objects whilst maintaining loose coupling between them. This gives you a system that is flexible and extendable with loose coupling without breaking the OPEN-CLOSED PRINCIPLE [7] (page 57).

The purpose of this paper is to present a filtering mechanism to avoid propagating all updates to all observers. To help illustrate this, we may use an analogy to a news publisher within a niche market. The news publisher charges his subscribers on a per news update delivered basis. Suffice to say not all subscribers wish to receive all the niche market news updates, thus the publisher has decided to offer custom subscription packages, where one can subscribe and receive all news updates, or only headline / breaking news updates depending on the individual needs of the subscribers.

Offering this functionality introduces a new challenge, the GoF [1] state this as the ”Push or Pull” model. In the ”Pull” model the `Subject` merely issues a notification of change without providing any extra information to the `Observers`. This means the `Observers` themselves must discover what has changed, and whether it is relevant for them. Conversely in the ”Push” model the `Subject` ”pushes” extra data to the `Observers`. Essentially the `Observers` get the change information served directly to them parametrically[2].

The example of the news publisher being the `Subject`, whereas all the customers are the `Observers` shows how to place responsibility onto the `Subject`, This goes beyond just using a push model, in addition the `Subject` must deal with what is analogous to the ”subscription type” of the `Observer`. The reason for using this model is to avoid unneccessary memory usage if dealing with ”heavy-objects”. A different pattern that resolves similar issues locally in an application is the VIRTUAL PROXY [1] and the ”Lazy” family of patterns (LAZY INITIALIZATION [8], LAZY LOAD [9]) for datalayer to businesslayer retrieval.

If the `Subject` is a provider of large and/or complex objects this will naturally take up significant amounts of memory on the `Observer` devices and induce performance issues if transferred over a network. If these complex objects are not necessarily needed by the `Observers` then this is a waste of resources.

On mobile, resource limited devices, a design should strive to constrain the memory footprint and inter-process calls of an application to a minimum. The initial memory footprint is affected by how many classes are loaded at initialization; for instance, loading entire libraries such as `System.Graphics.*` is wasteful if you do not actually need all the classes in the package. Secondly, the number of objects initialized and allocated will affect the memory footprint. Thirdly, all method calls will incur some overhead; although this is barely noticeable in intra-process calls it may have an effect on inter-process calls. These should therefore be minimized.

[2] A variation of this is the ”Event Listener” pattern wherein an Observer when registering with the Subject passes in a reference to an object that implements a pre-agreed method signature. The method signature contains a subclass of an abstract Event Class as an in parameter. This way the Event information is pushed to the Observer. This approach is used extensively in the Java.AWT and Swing components.

The original pattern of the GoF is named OBSERVER, this pattern has been named INTELLIGENT SUBJECT to emphasize the dominating role played by the Subject.

Intent

Relieve the `Observer` of all duties and avoid unnecessary resource usage when dealing with solutions where data passing from `Subject` to `Observer` is costly and `Observers` have divergent update needs by extending the `Subject` and giving it added responsibility.

Problem

You are faced with multiple clients each with differing needs for updates. Their needs may be based on limited resources, etc. thus they may only require updates when changes have gone beyond a certain level, or *threshold*. These requirements are individual. How do you accommodate varying needs in update frequencies in clients but still make this transparent for both the *subject* and the *observers*?

Motivation

If we break down the previously defined phrase 'mobile knowledge workers', we can tentatively motivate the use of the pattern. *Mobile* devices may imply limitations on resources in terms of capabilities or performance, or due to the cost of use or portability. *Knowledge* implies that the *workers* are dependant on information in order to do their job; for example, a healthcare worker or train ticket collector. The workers must have timely information available in order to do their job correctly, or the results could be less than agreeable. However, depending on the accuracy needs of the application the data may not need to be updated constantly. For instance, not all `Observers`, whether they be applications or different `Clients`, may require the same accuracy or timeliness.

Applying our real-world analogy, a person who is a news subscriber may choose to subscribe to only the headline news *if the cost of the subscription is too high* for a full news subscription. Similarly, an application on a resource limited device could opt to only receive updates when the data has changed by a pre-defined amount if notifications are costly. Note that although the example of a mobile knowledge worker *motivates* the pattern, it does by no means limit the applicability of the pattern to resource limited devices of the domain of mobile information systems.

An example of this could be in a financial application wherein certain `Clients` (`Observers`) are so resource limited that they cannot receive updates too frequently seeing as the updates are costly. Thus they only desire updates when values change beyond a certain limit or threshold.

This implies that the INTELLIGENT SUBJECT pattern gives the `Subject` additional responsibilities. Due to the fact the `Subject` must actively handle which `Observers` are to receive notifications anytime the data changes.

Forces

You are dealing with situations where there is a real need to provide updated information to different observers with varying requirements to data freshness. The solution must be stable in its interface and easy to bind to for observers, but at the same time it must be flexible and capable of accommodating differing needs. This creates an overarching force of providing a static interface while allowing for dynamic behaviour.

Applicability

The INTELLIGENT SUBJECT pattern can be applied in the following scenarios:

- Use the pattern when a `Subject` object needs to notify a dynamic list of unknown `Observer` objects without inducing strong coupling between the objects *and the `Subject` must handle divergent update needs from the `Observers`.*
- You need to utilize a ”push” model[3], however the cost of passing the `eventData` is too high to justify it being passed when the value of the data is of no significance to the receiver; for instance, because the data value change is too fine. Frequently pushing the `eventData` will lead to unacceptable performance. The performance cost can reside with the `Subject` or the `Observers`, or even both. On the `Subject` side, the cost may be associated with network constraints; for instance, messages fail to reach the intended Object, forcing the `Subject` to resend the message even though the `eventData` in the message is of no interest to the `Observer`. Contrarily, the cost may reside with the `Observer` if the cost of processing the received `eventData` is high in terms of computational power (which on a battery-powered device would translate directly into draining the battery). In which case it would be preferable for the `Observer` to only receive updates that are relevant.
- The `Observers` need to do cascading updates to many different aspects / objects upon receiving `eventData`. This is costly. Avoid this by defining upfront limits/thresholds for when to receive updates.

Solution

Create a separate class that handles the notification to only those observers that require it based on their individual thresholds. The Subject does not know, nor does it care, which of the observers actually receive its updates. Similarly the Observers do not know whether there have been sent out updates that they have not received, they are only notified whenever a change happens that exceeds their personal threshold. The Observers are thus able to create their own universe of state, or sphere if you will; which is not intruded or “contaminated” with unnecessary or uninteresting data. The following sections present the solution in more detail, starting with a structural view. This is followed by a behavioural view and a presentation of the participants, before finally a code sample and implementation guideline is provided.

[3] Perhaps because the client devices do not have pull capabilities.

Structure

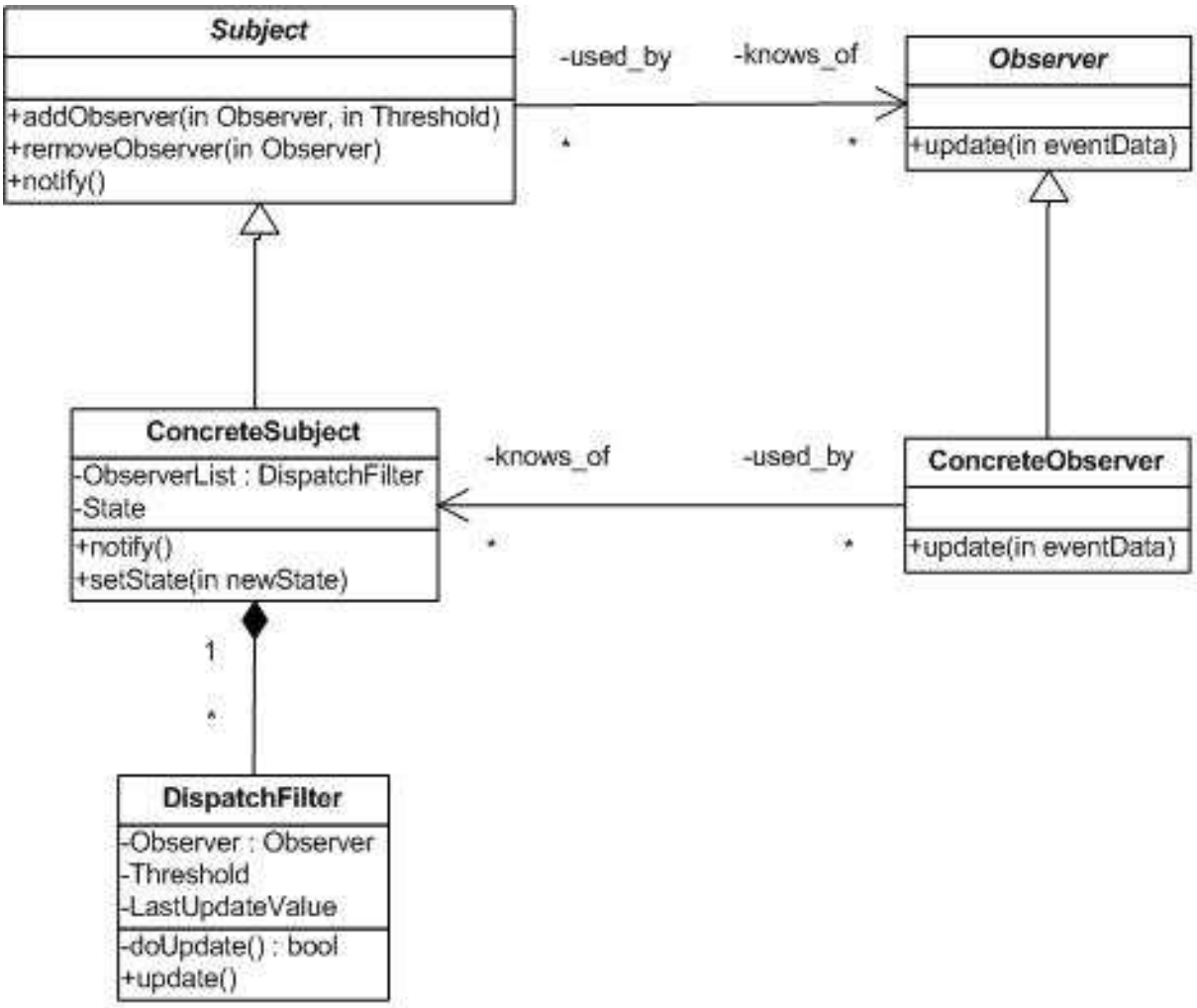


Figure 1: Class diagram of the INTELLIGENT SUBJECT.

The difference between the original OBSERVER pattern, and INTELLIGENT SUBJECT is in the addition of the `DispatchFilter` class. Though we previously stated that more responsibility is placed on the `Subject`, in that context, the term `Subject` was only considered conceptually. As we can see from the class diagram in figure 1, the pattern uses ”part-whole” composition because the `DispatchFilter` is contained within the `ConcreteSubject`. A filter could per se exist without the `ConcreteSubject`, however there would be little point in this since the `DispatchFilter` is uniquely associated with the `eventData` values of each individual `Subject`.

Another shift of responsibility is that the `ConcreteSubject` no longer invokes methods on the `Observers` directly. This task is delegated to the `DispatchFilters` contained in the `ObserverList` attribute. This accomplishes two things; the `ConcreteSubject` now has no knowledge of the specific needs of any `Observers`, nor should it. The `ConcreteSubject` knows only how many `Observers` are registered at any given time in its list, but that is all the knowledge it has. Additionally, this separation enforces the LAW OF DEMETER[4].

[4] Essentially, the law of Demeter states that a method *M* of object *O* may only invoke the methods of closely connected/related objects.

The separation of `DispatchFilter` into a separate class is crucial to avoiding DIVERGENT CHANGE, one of the many malodorous symptoms described by Fowler [10]. We are dealing with two distinct behaviors, lumping them both in with the `Subject` class is unattractive. The `Subject` class deals only with receiving notifications from the `Client` (assuming this is the notification model used), wrapping the whole event up in an `eventData` object, and notifying each of the members in the `ObserverList.` Only a reference to the `eventData` is passed to the `DispatchFilter` objects which save the load of a possibly large `eventData` object being unneccesarily transfer. This leads us on to the second behavior, namely the evaluation and propagation of `eventData` to the registered `Observers`. The task of evaluation is closely tied into the threshold values of the individual `Observers`, thus it should be performed by the `DispatchFilter` which is object that is composed of the `Observer` and threshold value. Additionally, the `DispatchFilter` must handle the computation of the difference between the new `eventData` value and the `lastValue` of the `Observer`. If the difference is greater than the threshold then an update will be initiated.

As a comment on Figure 1, it is plausible to create the object `observerList` class as a generically derived class parameterized with `<Filter>`, which the `Subject` class would then bind to. In such a case, we would be using an association between `DispatchFilter` and `Subject` instead of composition between `ConcreteSubject` and `DispatchFilter`. The advantages of this would be that a layer of indirection would be removed (the `ConcreteSubject` class) and we would enforce type-checking, and also ensure that the *addObserver* and *removeObserver* methods are correctly implemented by the `DispatchFilter` class. However, this is only applicable in certain strongly typed languages (although many languages do now support it with Java Generics and C# Templates), and does have consequences in terms of ”code bloat in [for example] C++” p. [11] .

The INTELLIGENT SUBJECT adaption of the original OBSERVER is reminiscent of the MEDIATOR pattern as described in the GoF book [1]. However, whereas the MEDIATOR pattern is concerned with centralized control of complex interactions between objects, in order to decrease the coupling between them, the INTELLIGENT SUBJECT is concerned with centralized control of divergent update needs. Concisely stated; MEDIATOR handles centralized control of cascaded / dependant updates, INTELLIGENT SUBJECT handles divergent update needs.

Participants

- **Subject**
  - knows of its `Observers` and offers all the method signatures needed by `Observers`, to add and remove themselves.
- **Observer**
  - Is an interface used by concrete `Subject` objects to update the registered `Observers`.
- **ConcreteSubject**
  - This participant extends the `Subject` base class. It also stores the state that is the basis for all `Observer` updates.
- **ConcreteObserver**
  - Knows of the `ConcreteSubject` so that it can attach itself and remove itself from the Subject’s list of `Observers`. Implements / Extends the `Observer` supertype to stay synchronized with the methods and signatures used for updates.

- **DispatchFilter**
  - This class is delegated the task of directly invoking the *update* method of all `Observers` where the `eventData` value exceeds their threshold value. It is also responsible for retrieving the state from the `ConcreteSubject`.

Collaborations

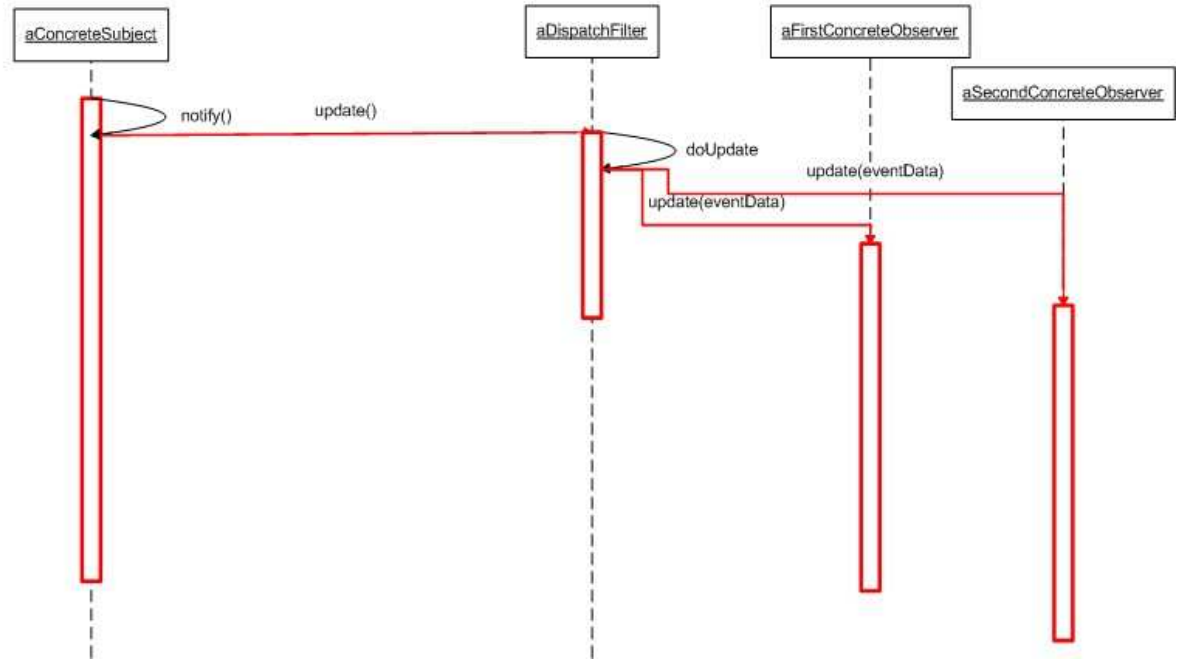


Figure 2: Sequence diagram of the object interactions during an update.

The sequence diagram shows the interactions between the objects during an update event. Initally the object `aConcreteSubject` will receive a message call to its *notify* (this is not shown in the above diagram). This will prompt the object `aConcreteSubject` to invoke its *iterateList* method. Essentially this is where the responsibility of the `concreteSubject` object ends. It invokes the *update* method on each `DispatchFilter` object in its `observerList`, it is then the `DispatchFilters` responsibility to decide whether or not the `Observer` object contained in the Filter object is to be updated with new `eventData`. This is depicted as the self-call of *doUpdate* the `aDispatchFilter` object-lifeline in figure 2. The *doUpdate* returns a boolean value after having compared the difference between the specific `Observer` object's `lastValue` attribute and the new `eventData` value against the `threshold` attribute value of that `Observer`. This is accomplished through operator overloading, since depending on the type of the `eventData` the operator symbols of greater than and less than may not natively be supported. In which case the `Observer` must overload those operators to function with the `eventData` type.

Code Sample and Implementation Guidelines

The following code samples show a C# skeleton implementation and the mechanisms behind the INTELLIGENT SUBJECT pattern.

Below is the *IObserver* interface which all classes that wish to observer the *Subject* must implement. For simplicity's sake, we are merely using an int object[5] as the eventData

[5] since the example is written in C# where all types ultimately derive from System.Object, as opposed to Java where int is an immutable primitve disconnected from the object model.

parameter that gets passed to the observers. Note that in a real implementation the eventData object would be more complex, and consequentially could be passed as a struct which is slightly more efficient as reported by [12].

```
public interface IObserver
{void Update( Object state );}
```

The class below is the *ConcreteObserver* which implements the *IObserver* interface.

```
public class ConcreteObserver : IObserver
{
   public void Update(Object state )
   { Console.WriteLine(id + " updated with " + state.ToString());}
   . . . .
}
```

The following (partial) abstract class is the Subject supertype, correlates to the Subject in the participant list. It also has methods for allowing observers to remove themselves and for notification.

```
public abstract class Subject
{
   private const int arrayno = 10;
   protected static DispatchFilter[] observerList = new
   DispatchFilter[arrayno];

   private int counter = 0;

   public void addObserver( IObserver observer, int threshold)
   {
         if(counter < arrayno)
         {
            observerList[counter] = new DispatchFilter(observer,
            threshold);
            counter++;
         }
         else
         {//throw an exception here}
   }
}



public class ConcreteSubject : Subject
{
   private int State;

   public void setState(int s )
   {
      this.State = s;
      notify();
   }


   private void notify( )
   {
         for (int arrayIterator = 0; j < counter; arrayIterator++)
         {observerList[arrayIterator ].update(ref State);}
   }
}
```

Listing 1: Code sample showing skeleton of the Intelligent Subject pattern.

The code snippet above, `ConcreteSubject` is the class which extends the abstract class `Subject`. As we can see it offers no method for `getState()` as the original OBSERVER pattern does, this is because INTELLIGENT SUBJECT enforces a push model, thus there is no need for Observers to be able to programmatically retrieve state since it is pushed to them as a parameter in the `Update` method.

Whenever the state is set, the `notify()` method is called. Note that a caveat about the sample above is that calling the `notify()` method sequentially with the state setting operation is not advisable. This is because in a real-life implementation the state setting procedures may be complex involving many steps, and multiple calls to the `setState()` method. Therefore the `Client` would not want to call the notification `notify()`until after the `setState()` method had been called for the last time. In practice, this is easy to implement; simply extract the call to `notify()` and place it in an overridden `notify()` method call. Thus the `Clients` could call `notify()` to run the updates. The only reason we didn't was to simplify the example.

Below in listing 2, the code that handles the filtering and `Update()` calls to observers is presented.

```
public class DispatchFilter
{
      private IObserver Observer;
      private Object Threshold;
      private Object LastUpdateValue;

      public Filter(IObserver observer, Object threshold) {
         this.Observer = observer; this.Threshold = threshold;
         LastUpdateValue = 0;
      }

      public void update(ref Object state ) {doUpdate(ref state);}

      private void doUpdate(ref Object state) {
         if (LastUpdateValue != 0 && beyondThreshold(state))
               Observer.Update(state);

         else if(LastUpdateValue == 0)
         {
               LastUpdateValue = state;
               Observer.Update(state);
         }
      }


      public bool beyondThreshold(Object state )  {
         return (getDifference(state) > Threshold);
      }

      public int getDifference(Object state ) {
       return state - LastUpdateValue; }
}
```

Listing 2: Code for the Filtering class which handles the *Update()* calls to the conditioned Observers

The `DispatchFilter` class encapsulates the behavior required to update the Observers and handle the task of filtering out which Observers are to receive updates. The `ConcreteSubject` class will invoke the `update()` method, of `DispatchFilter` objects maintained in its `ObserverList`, and pass in the `eventData` object (in our vanilla example this is just a simple Object) as a reference. Note that the `ConcreteSubject` *does this for all the* `DispatchFilter` *objects in its list*. It must be done this way to enforce the separation of concerns, and encourage the high cohesion of the Subject and DispatchFilter classes. Note that this approach (in certain languages) is not costly since passing `eventData` as a reference in-process is performance wise economical, and allows a higher cohesion in the `ConcreteSubject` class. This pass-by-reference approach is idiomatic to the C# programming language, and is also doable in C++, however it will not be possible in for example the Java language. In which case there might be a slight performance penalty, but in-process passing-by-value is not overly costly, so the message passing architecture is not bound to any specific programming languages.

The `DispatchFilter` object will then check whether the new `eventData` difference value exceeds the threshold of the individual `Observer`. If so, then the `Observer's Update()` method is called.

Note that this design lends itself well to Meyer's NON-REDUNDANCY PRINCIPLE [7] in the constructor of `DispatchFilter` and in the `addObserver()` method of `Subject`.

The NON-REDUNDANCY PRINCIPLE states that ”under no circumstance shall the body of a routine ever test for the routine’s precondition” [7] (page 343). We see that although the *addObserver* method in Listing1 does do a check on the size of the array, it does not do any checks on the integrity on the parametric data passed in. This affects the responsibility distribution, essentially the code sample operates with ”demanding pre-conditions” [7] (page 343).. The responsibility is to a larger degree shifted to the `Observer` which must ensure that any data passed when registering is correct, in this code sample if the data is not correct and the registration fails the `Observer` will not receive any notification of this. This approach goes against the paradigm of ”defensive programming”, and Meyer argues that the NON-REDUNDANCY PRINCIPLE allows for reduced complexity and increased reliability; this is called ”the zen-style paradox…: that to get *more* reliability the best policy is often to check *less*” [7] (page 345).

Consequences

A consequence of the INTELLIGENT SUBJECT pattern is the shift of responsibility between the `Subject`/`DispatchFilter` dyad. The `Subject` becomes a class that holds a list of all registered `Observers` under the guise of `DispatchFilter` objects, which handle the tasks of adding and removing `Observers`. However, the `Subject` no longer has the responsibility of communicating with `Observers` to Update them, this is now delegated to the `DispatchFilter` class. Compared to the original OBSERVER pattern, this variation is more complex as you use delegation to provide the filtering mechanism through a separate filter object. Additionally, there is transparency between the classes; for instance, the `Observers` do not know that there is a separate `DispatchFilter` class that updates them with new `eventData`. If they at times are bypassed, it is because the `eventData` change is below their threshold value, causing them to remain completely oblivious to the change. Therefore a chance of *data disalignment* between `Observers` can occur. This can be troublesome if the `Observers` in a different part of the system cooperate or collaborate and their data is not the same because they have different threshold values registered with the `Subject`. Thus they may have received a different number of updates, in which case one of the `Observers` would have more accurate and more timely data than the other. This could be solved by timestamping the `eventData` so that the `Observer` with the freshest data would trump the `Observer` with stale data.

Known uses

The traditional OBSERVER pattern (and minor variations on it) have been widely used in object-oriented event driven software designs. In the .Net and Java frameworks, the traditional OBSERVER pattern is utilized extensively in their delegate-event models [13, 14]. It is also used in the architecture of Symbian S60 platform for mobile devices [15]. The traditional OBSERVER has been used in the Java packages java.awt and javax.swing for handling notifications between graphical artefacts, event-triggers and the event listeners which handle the business logic.

The adoption presented here is viable in domains of resource limited devices, or in systems where any `eventData` is propagated over a network with limited bandwidth. Propagating this data to `Observers` who do not need it should be avoided. Concepts from the INTELLIGENT SUBJECT pattern have been used as part of Google’s Android Location API framework [16] wherein it is possible to register (an `Observer`) with a

`LocationProvider` (`Subject`) with a set `ProximityAlert` (conceptually a `threshold`), thus the `Observer` will not receive updates all the time, only when the proximity alert is triggered.

Related patterns

The original OBSERVER and INTELLIGENT SUBJECT are high-level design patterns. It would be feasible to use other patterns such as FACTORY METHOD to create Filters, or to use SINGLETON to ensure there is only one list object containing the Observers. As mentioned previously, the MEDIATOR pattern is similar to INTELLIGENT SUBJECT in its functional aims. Concisely stated, MEDIATOR handles centralized control of cascaded / dependant updates, whereas INTELLIGENT SUBJECT handles divergent update needs. The SASE OBSERVER [17] variation is similar, it allows the `Observers` to register with the subject, and at registration time identify themselves, register which events they care about, and register what event data they request when the event fires, and also possibly what they should do with the event data. Although very similar in many of its intents, the SASE pattern gives a different distribution of responsibility. It allows the `Subject` to dictate the `Observers` response and processing of events, whereas INTELLIGENT SUBJECT enforces an opaqueness between the `Subject` and its `Observers`; thus, the `Subject` does not know and does not care what happens to the data after it has been pushed to the `Observers`.

Another option is that the `DispatchFilter` could implement the STRATEGY pattern allowing for interchangeable algorithms to be applied to handle the filtering. Thus, the mechanisms that go into differentiating between which `Observers` receive updates could be run-time pluggable. This could allow the `DispatchFilter` to take on a policy-enforcer approach to notification, thus information could be disseminated not only based on which `Observers` that have registered for it, but also based on internal policies set forth by the `Subject` as to which `Observers` qualify as recipients (maybe based on the sensitivity of the information).

Finally, Niblett and Graham propose in the IBM Systems journal [18] a pattern called the NOTIFICATION PATTERN, also known as the SOA NOTIFICATION PATTERN. This pattern is manifested in the WS-Base Notification specification. The pattern is an alternative to INTELLIGENT SUBJECT as it allows for a `filter` to determine which `Observers` are to receive messages, thus not all notifications are propagated to all registered `Observers`. However the main difference is in INTELLIGENT SUBJECT dealing with compounded results, in the form of triggering thresholds as a mechanism for filtering. The NOTIFICATION PATTERN is more concerned with direct conditional limitations, such as topic based limitations. Furthermore INTELLIGENT SUBJECT is closer to the original OBSERVER [1] in that detachment can only be done by direct OBSERVER initiation, whereas NOTIFICATION PATTERN allows for temporal subscription based detachment so that an `Observer` may detach at a predefined time in the future.

Acknowledgements

I would like to thank my shepherd Maurice Rabb for his insights and contributions to improving this pattern. He provided strong technical advice, and many pointers to relevant materials, along with encouragement and support. I would also like to thank Jim Siddle for his oversight and suggestions, and also Andreas L. Opdahl for his comments on early versions of the paper. Finally I would like to thank my group at Euro-PLoP for their great feedback. .

References

1. Gamma, E., et al., *Design Patterns: Elements of Reusable Object-Oriented Software*. 1994: Addison-Wesley Professional. 416.
2. Christensen, C.M., J. Kjeldskov, and K.K. Rasmussen, *GeoHealth: a location-based service for nomadic home healthcare workers*, in *Proceedings of the 2007 conference of the computer-human interaction special interest group (CHISIG) of Australia on Computer-human interaction: design: activities, artifacts and environments*. 2007, ACM: Adelaide, Australia.
3. May, A., et al., *Opportunities and challenges for location aware computing in the construction industry*, in *Proceedings of the 7th international conference on Human computer interaction with mobile devices \& services*. 2005, ACM: Salzburg, Austria.
4. Patel, N. *Mobile World Congress Roundup: Cellphone Mania*. [Webpage] 2008 11/2-2008 at 10:16pm [cited 2008 11/2-2008]; Webpage summarizing all the cellphones released during the first week of Mobile World Congress 3GSM in Barcelona]. Available from: http://www.engadget.com/2008/02/11/mobile-world-congress-roundup-cellphone-mania/.
5. Mahmoud, Q.H. *J2ME and Location-Based Services*. 2004 [cited 2008 10/1-2008]; Available from: http://developers.sun.com/mobility/apis/articles/location/.
6. Android, G. *Location-based Service APIs*. 2007 [cited 2008 10/1]; Available from: http://code.google.com/android/reference/android/location/LocationManager.html.
7. Meyer, B., *Object-Oriented Software Construction*. 2nd Edition ed. 1997, Upper Saddle River, New Jersey: Prentice Hall PTR. 1254.
8. Beck, K., *Smalltalk Best Practice Patterns*. 1996: Prentice Hall PTR. 240.
9. Fowler, M., *Patterns of Enterprise Application Architecture*. 11th printing ed. The Addison-Wesley Signature Series. 2003, Boston: Pearson Education. 530.
10. Fowler, M., *Refactoring: Improving the design of existing code*. Object Technology Series, ed. J. Booch, Rumbaugh. 1999, Reading, Massachusettes: Addison-Wesley. 431.
11. Fowler, M. and K. Scott, *UML Distilled Second Edition: A Brief Guide to the Standard Object Modeling Language*. Second Edition ed. Object Technology Series, ed. J. Booch, Rumbaugh. 2000: Addison-Wesley.
12. Kayun, C. and S.-a. Chonawat, *Energy conscious factory method design pattern for mobile devices with C\# and intermediate language*, in *Proceedings of the 3rd international conference on Mobile technology, applications \& systems*. 2006, ACM: Bangkok, Thailand.
13. Richter, J. *An Introduction to Delegates*. MSDN Magazine The Microsoft Journal for Developers 2004 [cited 2008 22/1]; Available from: http://msdn.microsoft.com/msdnmag/issues/01/04/net/.
14. Microsoft. *Implementing Observer in .NET*. Microsoft Patterns and Practices 2003 [cited 2008 20/1]; Available from: http://msdn2.microsoft.com/en-us/library/ms998543.aspx.
15. Developer Community Wiki, N. *Design Patterns in Symbian*. Developer Community Wiki 2008 [cited 2008 15/1]; Available from: http://wiki.forum.nokia.com/index.php/Design_Patterns_in_Symbian#Observer_Pattern.
16. Google. *Google Android android.location.LocationManager*. 2007 [cited 2008 22nd of March 2008]; Available from: http://code.google.com/android/reference/android/location/LocationManager.html.
17. Brown, K. *Understanding inter-layer communication with the Self-Addressed Stamped Envelope (SASE) pattern*. [Webpage] 1998 18th of March 1998 [cited 2008 27/3-

2008]; Available from: http://members.aol.com/kgb1001001/Articles/SASE/sase2.htm.
18. Niblett, P. and S. Graham, *Events and service-oriented architecture: The OASIS Web Services Notification specifications*. IBM Systems Journal, 2005. **44**(4): p. 17.

Advanced Synchronization Patterns for Process-Driven and Service-Oriented Architectures

Carsten Hentrich

CSC Deutschland Solutions GmbH
Abraham-Lincoln-Park 1
65189 Wiesbaden, Germany
chentrich@csc.com

Uwe Zdun

Distributed Systems Group
Information Systems Institute
Vienna University of Technology
Argentinierstrasse 8/184-1
A-1040 Vienna, Austria
zdun@infoysys.tuwien.ac.at

In a process-driven and service-oriented architecture, parallel and independently running business processes might need to be synchronised according to dependencies of complex business scenarios. In this paper we describe three software patterns that address such rather advanced synchronisation issues between business processes running in parallel. The patterns focus on coordinating the parallel and principally independent business processes via architectural solutions allowing architects to model the flexible synchronisation of the processes.

Introduction

Service-oriented architectures (SOA) are an architectural concept in which all functions, or services, are defined using a description language and have invokable, platform-independent interfaces [Channabasavaiah 2003 et al., Barry 2003]. In many cases services are called to perform business processes. Each service is the endpoint of a connection, which can be used to access the service, and each interaction is independent of each and every other interaction. Communication among services can involve simple invocations and data passing, or complex activities of two or more services. In a *process-driven SOA* the services describe the operations that can be performed in the system. The process flow orchestrates the services via different activities. The operations executed by activities in a process flow thus correspond to service invocations. The process flow is executed by the process engine.

A *process* is a behavioural model expressed in a process modelling language, such as BPEL, that is instantiated and managed by a *process engine*. On a process engine multiple *process instances*

Proceedings of the 13th European Conference on Pattern Languages of Programs (EuroPLoP 2008), edited by Till Schümmer and Allan Kelly, ISSN 1613-0073 <issn-1613-0073.html>.

of one or more processes are typically running in parallel. Processes usually work on business data that is stored in *business objects*. Usually each process instance has its own private data space of business objects it creates, in order to limited problems of concurrent data access, such as data inconsistencies, deadlocks, or unnecessary locking overhead.

In this paper we describe three patterns that address advanced synchronization issues of parallel business processes. In this context *synchronization* means that execution in terms of the progression through the different activities of a process needs to be synchronized with other business processes running in parallel. The synchronization issues reflect requirements of complex business scenarios, and the synchronization dependencies cannot be modeled directly in the business processes via static control flow dependencies. As a result, conflicting forces arise due to the need for loosely coupling the synchronization concerns with the business process models. Besides technical forces, such as the problems of concurrent data access, supporting business agility is central. Business processes are subject to constant change. Hence, any suitable synchronization mechanism must be loosely coupled in order to support changes in the connected business processes.

The three patterns presented are:

- The REGISTER FOR ACTION pattern describes how to synchronize the execution of functionality of a target system with business processes running in parallel.
- The BUNDLE PROCESS AGENT pattern describes how business objects being generated by different parallel processes can be processed in a consolidated way.
- The PROCESS CONDUCTOR pattern addresses how to enable designers to model dependencies between parallel business processes by flexible orchestration rules.

Consider a simple example to illustrate the use of the patterns: various business processes of a company require contacting the customer via phone, but the company wants to avoid contacting the customer too often in a specific period of time. Hence, the phone calls should be consolidated. In such business scenarios that require synchronization of multiple process instances, the patterns described in this paper can be applied.

If only a specific *action*, like “put phone call into a task list” needs to be performed after synchronization has taken place, the REGISTER FOR ACTION pattern should be applied. However, the phone call might also require a business process preparing the phone call and this business process then usually needs access the private business objects of the synchronized processes. In this more complex scenario, the BUNDLE PROCESS AGENT pattern can be applied. Finally, if the need for synchronizing occurs within the processes and requires each of the processes to be stopped until the synchronizing action (which might be yet another business process) has happened, then PROCESS CONDUCTOR is applicable.

This simple scenario should illustrate: which of the patterns is chosen depends on the design of the business processes that need to be synchronized. In some scenarios, the patterns are mutual alternatives, in others combining them makes sense.

Register for Action

Business processes are executed on process engines. Sometimes the execution of an action is depending on the states of multiple business process instances running in parallel. When this is the case, the action can only be executed if those parallel business process instances have reached a certain state represented by a specific process activity.

As business processes are created and changed while the system is running, it is not advisable to define synchronization dependencies statically in the models. Instead it should be possible to define and evaluate the synchronization dependencies at runtime while still allowing business process to change independently.

How can multiple, dynamically created process instances be synchronized with minimal communication overhead and still be kept independently changeable?

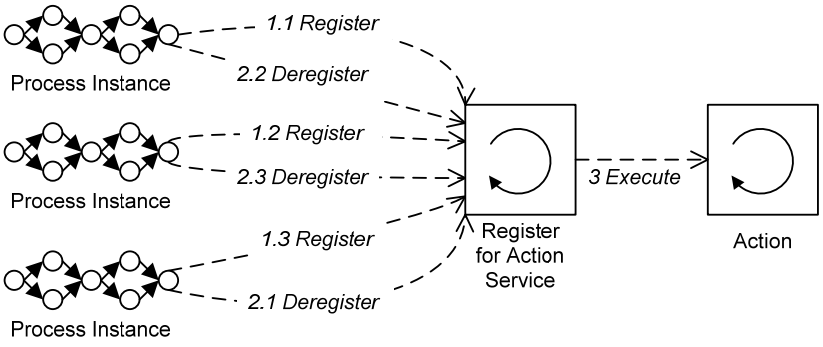
Business processes are dynamically instantiated on process engines and at different points in time. For this reason, there are usually several instances of one or more business processes running in parallel. Each of them has a different state of progression in terms of the process activity they have reached so far during execution.

When a specific *action*, such as a business function, service, or another business process, has a logical dependency to multiple business process instances, synchronization with all the process instances can be difficult. First of all, the action might not know which business process instances of the many possible parallel instances it is dependent on. But even if it knows the instances that it is dependent on, polling them for synchronization would incur a communication overhead. The same problem of a communication overhead for synchronization would also occur, if the process instances would run a synchronization protocol, for instance before triggering a callback that executes the action.

In addition, before the action can synchronize with the process instances, it needs to know that it must wait for one or more instances. That is, a mechanism is required to communicate that there is a new dependent process instance to wait for.

Business processes can change over time and new processes are constantly created, while the overall system is running. This includes that a state at which an action must be executed might change, gets added to a process, or gets removed from the process. The actions that are depending on business processes must be able to cope with such process changes. The effects of these changes should be minimized and should not impact other components in order to be manageable. A consequence is that the synchronization dependencies of the actions cannot be statically modeled in the models of the business processes or the actions, but must be defined and evaluated at runtime. In other words, a loose coupling between the action and the business processes it is dependent on is required.

Use a REGISTER FOR ACTION component that offers registration and de-registration services to be invoked from business processes. The registration informs the REGISTER FOR ACTION component to wait with the desired action to be initiated until a specific process instance has de-registered itself. When all registered processes have de-registered themselves the action will be executed.



The REGISTER FOR ACTION component offers two services:

- A registration service, where a process instance can register itself with its instance ID.
- A de-registration service that allows a process instance to de-register itself via its instance ID.

Invocation of the de-registration service means that the process has reached the state that is relevant for the action. The two services are invoked by process activities. Each registration invocation must have one corresponding de-registration invocation in a business process. This design has the consequence that the place of invocations can change as the business processes change over time. In other words, the REGISTER FOR ACTION component and the business processes are loosely coupled.

The REGISTER FOR ACTION component waits until all registered business processes have de-registered themselves. After the last de-registration, the action is executed by the REGISTER FOR ACTION component.

An important detail of a REGISTER FOR ACTION design is to determine the point in time when registration ends. As most scenarios of the pattern concern long running business processes, a registration delay is a practical solution that works in many cases. The registration delay runs a certain amount of time from the point in time when the first registration to the REGISTER FOR ACTION instance happens. For instance, if a registration delay of one day is chosen, then all registrations that accumulate throughout that day will be included. Of course, the length of the delay can be adjusted based either on previous experiences and experimentation. An alternative to a registration delay is introducing a specific registration type that ends the registration process for one REGISTER FOR ACTION instance.

Modeling the de-registration service invocation might be an issue for some business processes: de-registration should often be invoked as early as possible in order not to produce unnecessary delays for the action to be executed. If the business process contains complex decision logic there may be various paths that may lead to a de-registration service invocation at many different positions in the process. As the process execution may follow only one case specific path, de-registration must be found on all possible paths if a registration has been previously performed.

To place the de-registration service invocations at the right positions and to avoid multiple invocations of the de-registration service in case of loops in the process is sometimes not trivial, if the business process is of higher complexity. The easiest way might be to put de-registration simply at the end of the business process and thus to avoid the possible complex logic that is initiated by the different possible paths or loops. However, this is not always possible or optimal if de-registration as early as possible is required.

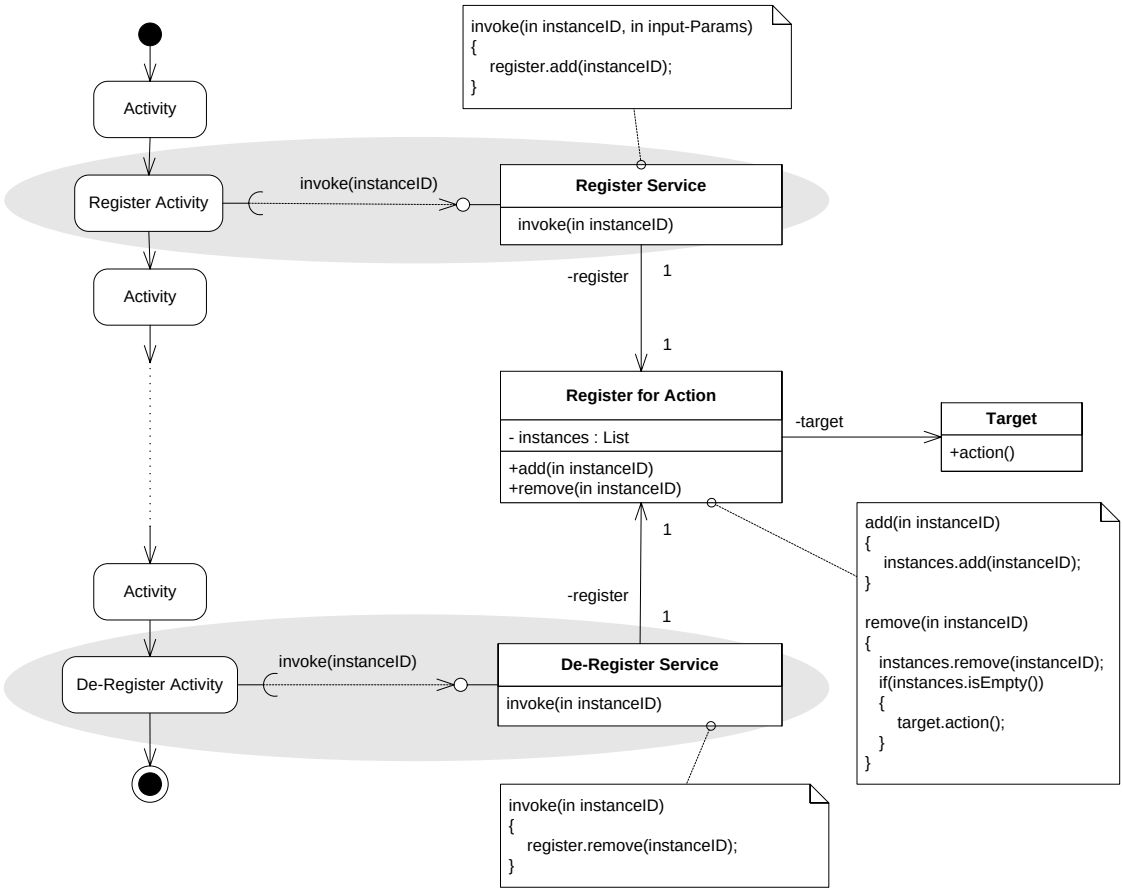
The service invocations from the business processes might be realized as SYNCHRONOUS SERVICE ACTIVITIES or FIRE AND FORGET SERVICE ACTIVITIES [Hentrich et al. 2008]. The realization using SYNCHRONOUS SERVICE ACTIVITIES is usually better suited as it is important for the business process to get informed whether the registration and de-registration was successful. If the target action is related to a more complex business process, then this consolidation can be achieved by using a BUNDLE PROCESS AGENT.

The ACTIVATOR pattern [Schmidt et al. 2000] has a similar structure as REGISTER FOR ACTION. It, however, solves a different problem, the on-demand activation and deactivation of service execution contexts to run services accessed by many clients without consuming resources unnecessarily. The patterns can be used together with REGISTER FOR ACTION using a shared structure. That is, the registration and deregistration services could be used for on-demand activation and deactivation.

Also, PUBLISH/SUBSCRIBE [Buschmann et al. 2000] has a similar structure, as it includes registration/deregistration of publishers and subscribers. This pattern can also be combined with REGISTER FOR ACTION using a shared structure. That is, the registration and deregistration services could be used to subscribing and unsubscribing to events for the time of being registered. This way, the REGISTER FOR ACTION component can communicate with its currently registered processes.

Example: REGISTER FOR ACTION example configuration

The following figure shows on the left-hand side a business process that invokes a register service. At the end of the business process, an activity invokes a deregister service. Both services are interfaces to a REGISTER FOR ACTION component, which maintains an instance list. When all instances are removed from this list, the action is invoked.



Example: Saving Costs of Postal Mail Sending

In the context of business processes that create information that must be send by mail to recipients, the pattern provides significant potential to save postal costs. If each business process produces its own letters to be sent to recipients a lot of postal costs will be created. It would be better to gather all the information created from the business processes and to wrap them in one letter. The idea of sending fewer letters will save significant postal costs. However, the problem is how to gather all the information and when is the point in time to pack all the gathered information in one letter and send it out to a recipient.

Applying the REGISTER FOR ACTION pattern it is possible to control and coordinate sending out letters to various recipients. The action associated in this context is sending a letter out to a recipient. This can be coordinated by registering all business processes that will create information to be packed in one letter for a specific recipient. Thus, the registration service can be designed to capture an additional parameter to specify the recipient. That way it is possible to pack all the information created for one recipient on one letter as the letter will be sent out when all registered business processes have de-registered themselves for a recipient.

The logic associated to the registration service and the REGISTER FOR ACTION component might be even more complex, e.g. to distinguish different priorities for the information to be sent out quickly or to send it out later. As a result there may be more complex rules to control and synchronize the business processes. However, the basic pattern represented by REGISTER FOR ACTION will always be the same.

Especially in a customer order or service management context the pattern is useful for these kinds of purposes, as to control communication towards recipients. It is suitable not only for postal communication but for various communication channels, e.g. fax, e-mail, or even telephone. The general purpose of gathering relevant information first, before initiating the communication, generally applies to all those channels.

Even as far as the telephone communication is concerned, it becomes clear that calling the recipients only once to discuss a whole bunch of open questions that stem from different parallel clearing business processes, for instance, will be better for customer satisfaction than contacting the customer several times to clarify one single issue at a time, which might even be of minor importance. The pattern provides flexible means to capture all these business scenarios and to automate significant parts of the business logic.

Known Uses

- The pattern has been used in projects to control batch processing of larger transactions. Each business process generates transactions to be made but the actual commit of the all gathered transactions needs to be done at a point in time when all related transactions to be made are identified. That way transaction costs can be saved by putting related transactions, addressed to the same account, in one larger transaction.
- The pattern has also been used to control the point in time when consolidated outbound communication to one concrete party needs to be performed in an order management context.
- The above purposes of the pattern have been used in projects in the telecoms industry in the context of order management, in the insurance industry in the context of claims handling. The pattern has also been used in banking as far as the mentioned transaction processing issues are concerned.

Bundle Process Agent

Business processes are executed on a process engine, and during their execution business objects are created and manipulated by the business process instances. Each business process instance creates and manages its own set of business objects in order to avoid data inconsistencies, locking overheads, deadlocks, and other problems created by concurrent data access. Business scenarios, such as consolidated sending of postal mail in batches, require consolidating the business objects being generated by many different process instances and then process them using a central but parallel running business process. Hence, the usual mechanisms of a process engine, in which each process instance keeps its business objects in a private data space, are not sufficient to support such scenarios.

How to gather the business objects from various business process instances and process them in a consolidated way without causing unwanted effects of concurrent data access?

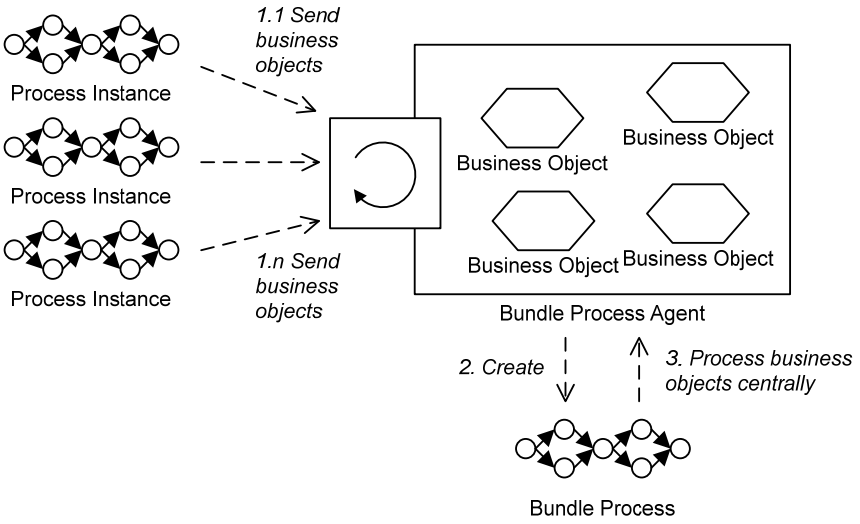
Business process instances running on a process engine have their own data space and are thus disjoint entities. When business objects are created during the execution of a business process, only the business process instances creating the objects know about their existence. That is, the business objects created by a business process instance are per default private to the business process instance. This helps to avoid unwanted effects, such as data inconsistencies, locking overheads, or deadlocks, when business process instances are running in parallel because the actions of the business process instances control all accesses to these business objects.

This technical concept can be applied to implement most business scenarios. However, there is a special case, where this technical solution does not work well alone: Consider that the business objects created by many different parallel business process instances are input objects to be processed by a central business process that logically gathers all these business objects and then processes the consolidated set of business objects. A typical example is that the business requires this business objects to be handled in a consolidated way, such as sending one letter per postal mail for a number of parallel transactions with a customer, instead of sending multiple letters. In this case, the parallel running consolidation process instance must gather the objects and process them. Unfortunately, usually process engines do not directly support such scenarios.

It is necessary to only centrally process those business objects that actually should be processed in a consolidated way. It might be that this is only a subset of business objects owned by a process instance. A process instance should still have control what business objects should be processed in a consolidated way and should thus be able to publish only those objects that it considers to be relevant.

Each of the involved business processes can potentially change over time. Hence, the consolidation architecture should not impose restrictions on the business process design that would hinder rapid changeability.

Send the business objects to be processed centrally to a BUNDLE PROCESS AGENT via a dedicated service, specified in the model of the business process. The BUNDLE PROCESS AGENT creates an instance of a bundle (consolidation) process, if no instance exists yet, and for each bundle it makes sure that only one instance of the consolidation process is running at a time. The business object bundle is gathered from different business processes invoking that dedicated service for sending the business objects. When a specified end criterion is reached, such as a deadline or a specified number of business objects in the bundle, then the bundle is centrally processed by the bundle process.



Design an architectural component that serves as a BUNDLE PROCESS AGENT, which offers a service to be invoked by business processes to send business objects that need to be processed centrally. The BUNDLE PROCESS AGENT stores the business objects being sent to it in a container that serves as a temporary repository. The container is not intended as the actual persistence mechanism of the business objects—it is rather intended to capture only what objects need to be processed centrally.

For this reason, this container might only keep BUSINESS OBJECT REFERENCES [Hentrich 2004] rather than the business objects themselves. However, it is also possible to send copies of actual business objects and not just references. Often these objects then only contain a subset of the business data of the original business objects, i.e. the subset of data that is relevant for processing the business objects centrally. In this case, it is advisable to introduce special types of business objects designed for these purposes.

The BUNDLE PROCESS AGENT waits until a specified end criterion is reached. For instance, this can be a deadline or specified maximum amount of business objects that can be bundled in one bundle. When the end of bundling is reached, the BUNDLE PROCESS AGENT instantiates a *bundle process* that processes the business objects centrally. The container with the business objects is cleared

after the processing has been initiated to be ready to store new objects for the next iteration. Only one instance of the bundle process is running at a time for each bundle, i.e. the processing of a set of business objects must be finished before the next instance of a bundle process can be started. Of course, different bundles can be assembled in parallel. Consider for instance, business objects for postal mail communication with customers are bundled, to send them together. Then there is one bundle per customer.

During the execution of the bundle process new business objects are sent to the BUNDLE PROCESS AGENT by business processes running in parallel for the next iteration. These objects are again stored in the container. This way, only business objects relevant for the next iteration are kept in the container, as the container is emptied when a new iteration, i.e. a new instance of the bundle process, has been started. The BUNDLE PROCESS AGENT repeats this process in a loop.

The BUNDLE PROCESS AGENT is implemented as a COMPONENT CONFIGURATOR [Schmidt et al. 2000] to allow controlled configuration of the agent at runtime. When it is initialized it performs the following functions:

1. It is checked whether there are new business objects in the container to be processed by a bundle process
2. If there are new objects it checks whether an instance of the bundle process is still running. Only if there is no instance running, a new instance is created that processes the new objects in the container. The container is cleared to be empty for new objects after the process instance has been started. If an existing instance is still running then no action is performed, i.e. no new bundle process is created nor is the business object container being emptied.
3. The agent loops back to step 1 until the loop is aborted by some event to finalize the execution or to suspend the execution.

There is one concurrency issue involved in this algorithm. The service that allows business processes to send new business objects to the container might conflict with the clearing action of the container that is initiated by the algorithm described above. That means a new instance of the bundle process might be created with the given objects in the container at that point in time. After the instance is created, the algorithm prescribes to clear the container. If there are new objects added to the container while the creation of the new instance is still in progress, then these objects will be deleted from the container with the clearing action without being processed by the bundle process. In order to avoid such a situation the container must provide locking and unlocking functions that are used for the time a new instance of a bundle process is created.

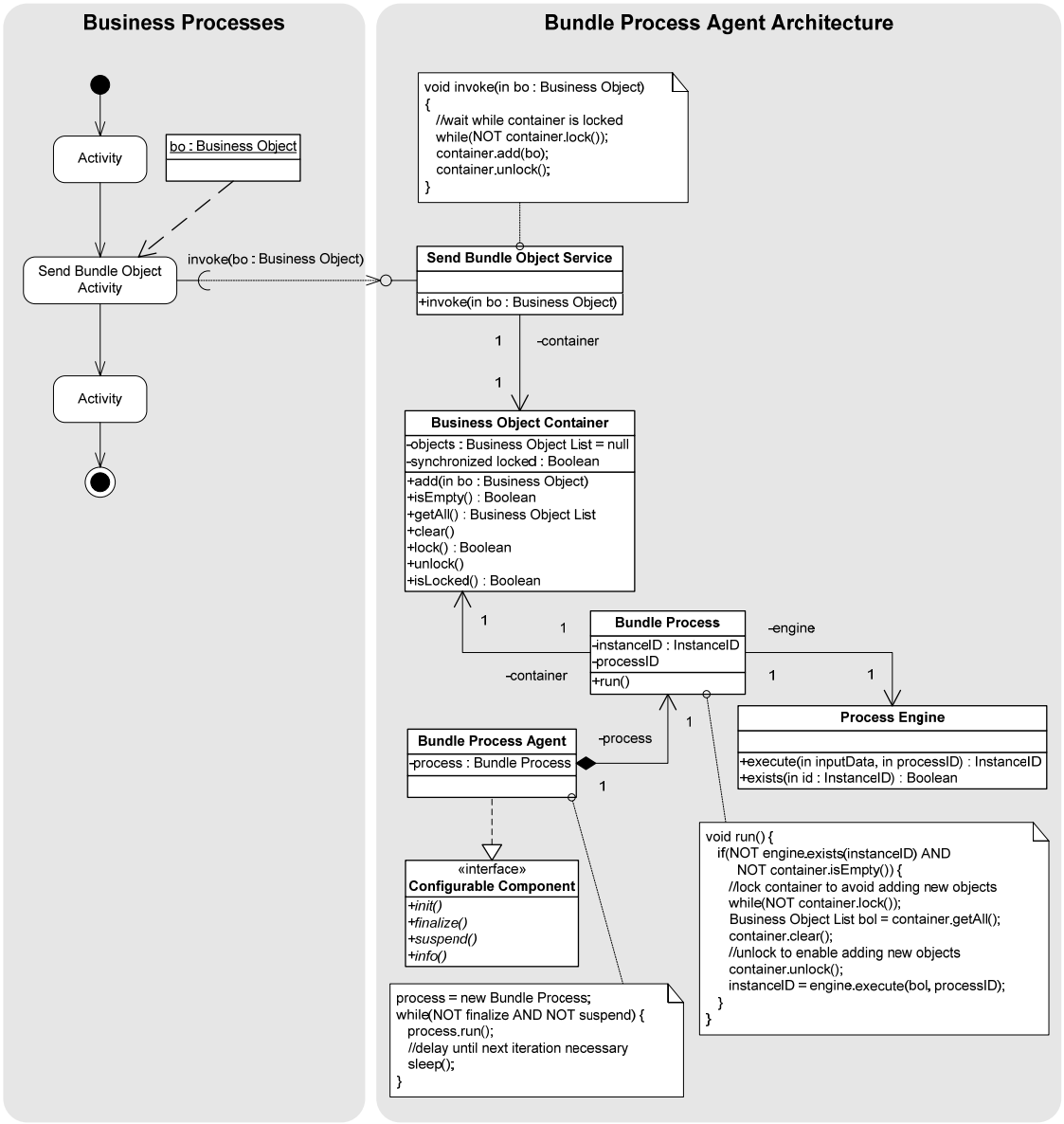
The BUNDLE PROCESS AGENT pattern thus resolves issues according to complex bundling of business objects that need to be centrally processed and offers a general architectural solution that is both flexible and extensible. Different bundle processes can be used for different purposes, though this will increase the complexity of the architecture. However, there might be larger effort involved to design a BUNDLE PROCESS AGENT component. For this reason, the pattern may only be suitable in projects and programs that have a larger strategic perspective.

The pattern can be combined with the REGISTER FOR ACTION pattern in order to dynamically identify what business processes need to be considered by the bundling. As far as the service invocation from business processes is concerned the SYNCHRONOUS SERVICE ACTIVITY pattern or the variation of the FIRE AND FORGET SERVICE ACTIVITY including acknowledgement [Hentrich et al. 2008]

is usually recommended to achieve some level of security that the business objects being sent have arrived at the initiated target. According to the MACRO-MICROFLOW pattern [Hentrich et al. 2007] the bundle process can be implemented as a macroflow or microflow.

Example: BUNDLE PROCESS AGENT example configuration

The following figure provides an overview of the conceptual structure how the BUNDLE PROCESS AGENT might look like including the service that provides the functionality to add business objects to the container. The structure also resolves the described concurrency issues by providing locking functionality of the container.



The figure shows business processes that invoke a special *Send Bundle Object Service* to send business objects. The processes may run in parallel and the services might be invoked at different points in time in the processes, i.e. the service invocation might be modeled several times in one process and might be used in various process models. The service simply adds the objects to the container. To resolve the concurrency issue, explained above, it uses locking and unlocking mechanisms. The class *Bundle Process Agent* implements the COMPONENT CONFIGURATOR [Schmidt et al. 2000] and invokes the *run* method of class *Bundle Process* in a loop. The *run* method of class *Bundle Process* retrieves the business objects from the container and creates a new bundle process if no instance is running and the container is not empty. It also uses the locking and unlocking mechanisms to prevent the concurrency issue.

The class *Process Engine* provides an interface to the API of the process engine being used to implement the business processes—in this case especially the bundle process. The *execute* method instantiates a bundle process with the given input data, which are the business objects from the container in this case. The *exists* method allows to check whether an instance of the bundle process, identified by a unique ID, is still running in the engine.

Example: Handling Complex Orders

The following example shows how two distinct strategic goals (mentioned in the known uses) have been realized using the pattern. In one larger project in the telecoms industry two important issues occurred in the context of processing complex orders from larger customers. Complex orders consist of a number of sub-orders that are processed in parallel business processes by different organizational units in the telecoms company. These sub-orders are independent to a certain degree from an internal perspective of the company. For this reason, the business processes for these sub-orders run in parallel to speed-up the completion of the overall order. However, in order to improve customer satisfaction and to reduce costs, issues that occur during the processing of sub-orders need to be clarified with the customer, whose perspective is on the overall order.

If each business process is implemented with its own issue resolution process the customer needs to be contacted for each single issue that might occur in a sub-order, or issue resolution might only be structured according to sub-orders. As a result, each process needs to implement its own issue resolution procedure in some way. To reduce the number of customer interactions and to save communication costs, the issue clarification process needs to be consolidated and treated as an own concern. That way, different processes can use the same clarification procedure and changes to these business processes associated to processing of sub-orders can remain independent.

Moreover, each customer has its own communication preferences, i.e. some want to be contacted by letter, others prefer e-mail or fax, and other customers rather prefer direct telephone communication. Additionally, some serious issues required written communication. Consequently, it was required to treat issue resolution as an own concern and to centralize the rules around the communication preferences. A concept for classification of occurring issues and a central processing of those occurring issues was required. The actual issue clarification process needed to be implemented as a rather complex business process itself that gathers all the occurring issues from those various parallel running sub-order processes. The rather complex rules for communication needed to be implemented by the issue clarification process.

The BUNDLE PROCESS AGENT pattern has been applied to deal with these requirements. A classification scheme for possible occurring issues has been designed in a business object model. The parallel running sub-order processes have just sent a type of issue that occurred during the process to the BUNDLE PROCESS AGENT. The agent created an issue clarification process that processed the issues centrally according to communication preferences. For instance, a list of issues that resulted from various sub-orders could thus be clarified in a single telephone call with the customer, or have been communication in a single letter. Direct communication via telephone of a consolidated list of issues has thus speeded-up the clarification process or has saved mailing costs, as a number of relevant issues have been gathered in a single letter.

The overall clarification process could be implemented in a controlled way considering the customer view and preferences of the overall order, while still having the ability to process the sub-orders according to different specialized internal departments. A special team to improve the clarification process as a separate concern could thus be implemented without affecting actual order processes. In that way, it has been possible to design the business process models according to different levels of expertise and to assign dedicated resources with expertise on issue resolution.

The issue classification scheme via the special business object model and the service for sending occurring issues via a service provided a clear interface that allowed new or improved sub-order processes to use it in a flexible way. The service has provided a defined interface for handing issues for clarification in a universal way. Customer satisfaction has been improved by classification of the issues and reducing the number of necessary interactions with the customers.

Known Uses

- The pattern has been used in several projects for the purpose of consolidating outbound communication to the same party in order to save costs by putting the communication content resulting from various business processes running in parallel. in one bundled communication. The bundle process controls the actual generation of the communication including format and media, i.e. letter, e-mail, fax, or even telephone, and the procedure to control the outcome of the communication. The purpose in the context was also to improve customer satisfaction via the consolidation of the communication via reducing the number of interactions and applying preferred communication mechanisms.
- The pattern has also been used to gather issues to be clarified with customers that result from various business processes running in parallel associated to a complex order. The issues are collected first and are then clarified with the customer rather than discussing each issue separately with the customer. That way issues could be clarified in relation to each other. The bundle process controls the clarification procedure in terms of a dedicated business process.
- The above two purposes of the pattern have been used in the context of order management in the telecoms industry and in the context of claims handling in the insurance industry. The pattern has served in this context as an architectural solution to the consolidation issues mentioned in larger strategic architecture projects.

Process Conductor

Interdependent processes are in execution on a process engine. The interdependency implies that execution of the processes needs to be synchronized at runtime. When business processes are executed on a process engine there are often points in these processes where dependencies to other processes need to be considered. That means a process may only move to a certain state, but can only move on if other parallel processes have also reached a certain state. Further execution of these parallel running processes need to be orchestrated at runtime, as it cannot be decided at modeling time when these states are reached or what these states are due to the separate execution of the processes and the fact that each process is a component that may change individually over time. In most cases, the rules for the orchestration need to be flexibly adaptable.

How can the interdependencies between processes be flexibly captured without tightly coupling the processes?

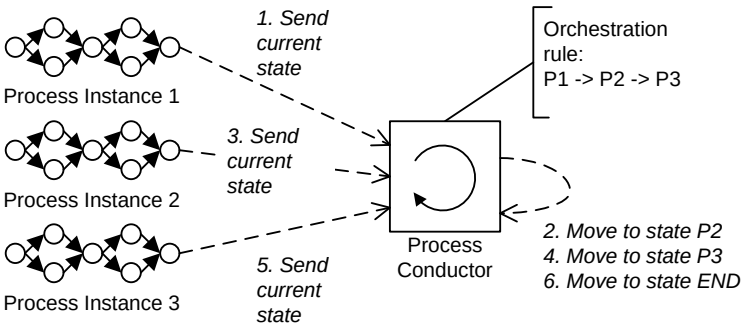
At some point in a process it might be necessary that the process is only allowed to move on if other processes have also reached a certain state. However, each individual process does not know what these states of other processes are and when they are actually reached, as each process runs at its own speed within its own data space. Moreover, each business process needs to be treated as a separate component and may change individually over time. Thus, processes need to be very loosely coupled as far as this aspect is concerned.

The reason for this is that it is very hard to specify in a single process model what states of other processes are relevant from an individual process's point of view and when these states are reached, nor what the relevant dependent processes are. If this is statically modeled in a process somehow the implementation will be very inflexible and changes to orchestration rules usually impact all involved processes. That means, the actual orchestration appears to be a complex concern of its own and the rules for orchestration cannot be defined attached to an individual process model. Consequently, the orchestration rules should not be captured as some types of static relationships of a process to other processes. The dependencies that will be generated if each process should know the rules for orchestration will be very hard to manage. If the rules change then each individual process needs to be changed as well. For this reason a tight coupling of the rules to each individual process is an inappropriate approach.

As a result, each process needs to be treated as an encapsulated component that does not know anything about the orchestration rules or the processes that it has dependencies to. Each process must only know its own points in the process where the dependency occurs but not what this dependency is about. New processes might be created over time, which create new dependencies and this must not affect existing process models to make the changes manageable. Each process model itself may also change, e.g. new steps are added without affecting the actual rules for orchestration as they need to be treated as a separate concern. The very problem is thus that the processes are standing in dependency but must actually not know

very much about each other, as the dependency needs to be separated out of the process to treat it as a separate concern and to make the processes and the complexity generated by these dependencies manageable.

Introduce a PROCESS CONDUCTOR component that offers configurable orchestration rules to conduct a number of business process instances. The PROCESS CONDUCTOR offers a service that is only invoked synchronously by the business process instances. Each process instance provides its current state in terms of the activity it currently performs as an input parameter to this service. The service returns a result to a specific process instance only when the orchestration rules allow the process to move to the next step. This way the order of the process instances to proceed is determined via the orchestration rules of the PROCESS CONDUCTOR.



A central aspect of the PROCESS CONDUCTOR pattern is that the central conductor is only invoked synchronously. That is, when a business process reaches a critical state where it may only move on if certain other dependent processes have also reached a certain state, then a SYNCHRONOUS SERVICE ACTIVITY [Hentrich et al. 2008] is modeled at this point in the process that invokes a service. At this point, the process to be conducted blocks on the synchronous invocation until the conductor returns a result. The PROCESS CONDUCTOR service reports the state of a process instance and the ID of the instance to the PROCESS CONDUCTOR component. The states and corresponding process IDs are stored in a container. The PROCESS CONDUCTOR component applies orchestration rules which are configurable to determine the order of events that need to be fired to initiate dedicated process instances to move on.

The PROCESS CONDUCTOR applies its orchestration rules to the states and corresponding process IDs in the container. The orchestration rules simply define an order of the process states, i.e. an order of terminating the corresponding process activities. The conductor then fires events to the process instances identified by their IDs in the order that is determined by the orchestration rules. Hence, the service implementation to report the state and the process ID can be implemented as an EVENT-BASED ACTIVITY [Köllmann et al. 2007]. The process engine receives the events and terminates the activities in the order directed by the conductor. As a consequence

the processes move on to the next step in the right order. The conductor repeats this process in a loop, as new processes may have registered for the next iteration.

The triggers to start one iteration of this procedure to apply the orchestration rules and to fire events to the processes can be twofold. It can happen repeatedly in defined time intervals, or it can be initiated by other dedicated event triggers, e.g. a master process has invoked the service of the PROCESS CONDUCTOR to register an initiation state that triggers the orchestration rules.

The registration of the state and process ID and the waiting position for the actual termination event to occur can also be designed as an ASYNCHRONOUS RESULT SERVICE [Hentrich et al. 2008]. In this case the business process needs to model two activities: one that places the request and a second one that gets the result, i.e. waits for the termination event.

The EVENT-BASED PROCESS INSTANCE [Hentrich 2004] pattern can also be used in conjunction with a PROCESS CONDUCTOR in case it might take a long time until the termination event occurs and it makes sense to split-up the process in two parts. That means if the termination event occurs the second part of the process will be instantiated rather than modelling a waiting position.

One must note that the pattern generally assumes that the process engine processes the terminate events in the sequence that they are fired. This implies that the activities will terminate in the intended order, i.e. the order the terminate events have been fired, and the processes will correspondingly move on in the right order. If this cannot be assumed then it might be that the activities of the processes do not terminate in the right order and consequently the processes do not move on in the right order as well. To resolve this issue the implementation can be extended by an additional service that is invoked from a business process. This additional service confirms that the activity has terminated. This is modelled as a second SYNCHRONOUS SERVICE ACTIVITY right after the first one. The next process, according to the rules, is only notified after the confirmation from the preceding business process has been received.

This may only be necessary if the order of termination is important within one iteration of notification. In many cases this is not important as it is rather the whole iteration that represents the order, i.e. all processes of one iteration may literally move on at the same time and slight differences do not matter. This also depends on the underlying rules and what these rules are further based on. According to the MACRO-MICROFLOW [Hentrich et al. 2007] pattern this may also depend on whether we are at microflow or macroflow level. Transactional microflows usually run in much shorter time (sometimes milliseconds) and even slight time differences might matter while these slight time differences might not matter at all at the macroflow level.

The pattern provides a flexible solution to determine the order of process steps that need to be synchronized by configurable rules. New processes and rules can be added without changing the architecture. Existing rules can also be modified without changing the implementation of running business processes. However, this flexible concept requires additional design and implementation effort. The design might be quite complex depending on the requirements regarding the complexity of the synchronization. For this reason, the pattern is most suitable in larger projects where architecture evolution and business agility is required.

The PROCESS CONDUCTOR pattern is a central bottleneck and it also incurs the risk of deadlocks in case the PROCESS CONDUCTOR'S orchestration rules are misconfigured or a business process fails to signal its state. In such cases, usually manual intervention is required. It makes sense to monitor the PROCESS CONDUCTOR component for such events.

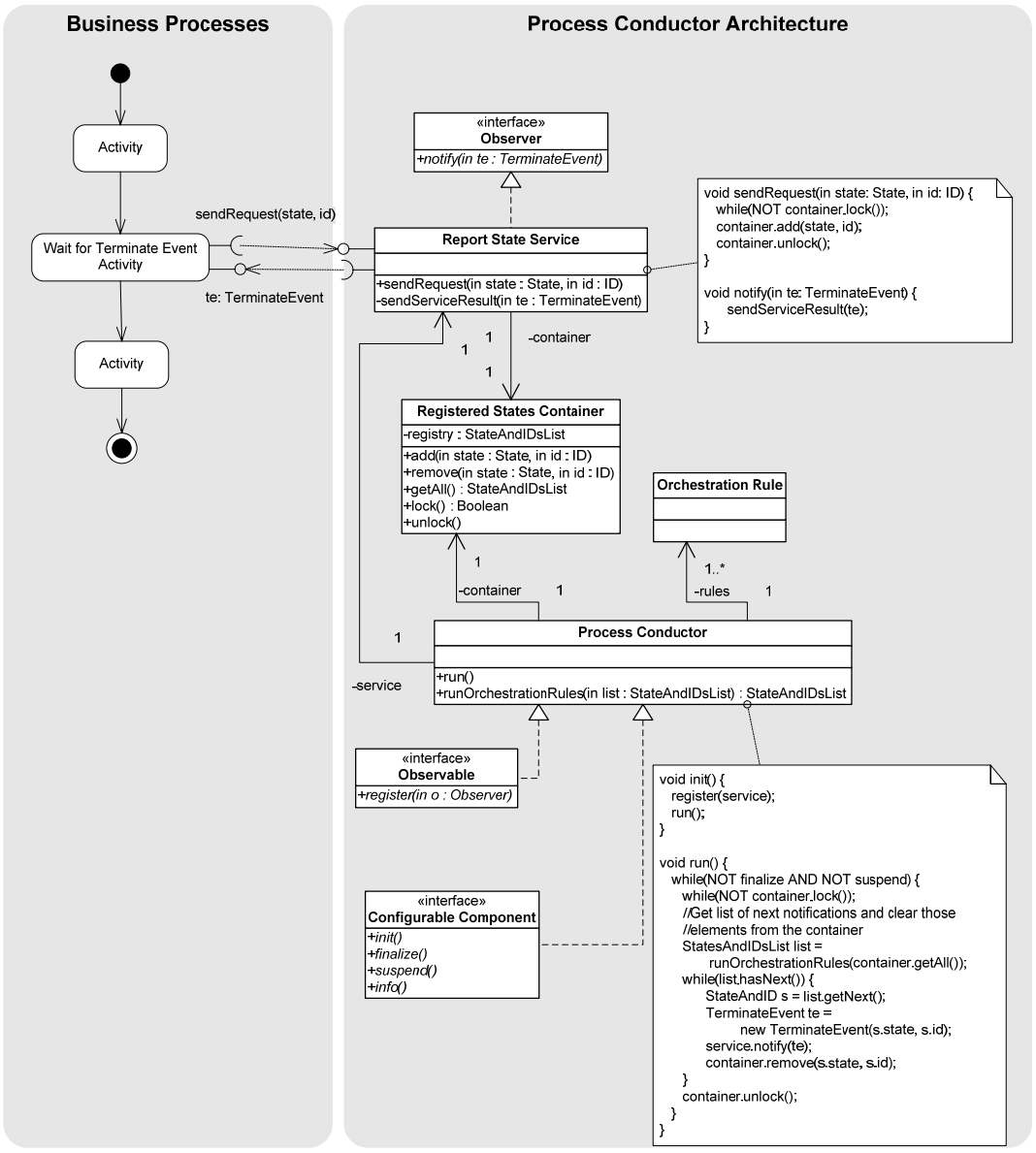
Example: PROCESS CONDUCTOR example configuration

The following figure shows an example of the general architectural concept of the solution using the OBSERVER pattern [Gamma et al. 1994] to notify the EVENT-BASED ACTIVITY of a process that waits for a terminate event to occur. The very order of sending these terminate events, i.e. the order of invoking the *notify* method of the observer class, is defined by the orchestration rules of the conductor. The orchestration rules just order a list of states associated to process instances and deliver those process instances that need to be informed in the next iteration. In the example in the figure this logic is hidden in the *runOrchestrationRules* method. That method takes a list of states associated to process instance IDs, runs the rules over them and delivers the list of process instances that apply to the rules for the next iteration. Those process instances are removed from the list given as an input parameter. The *Process Conductor* class itself is implemented using the COMPONENT CONFIGURATOR [Schmidt et al. 2000] pattern. In the figure, the class *Process Conductor* is the observable that is observed by a *Report State Service*. This service is invoked by activities from business processes as a SYNCHRONOUS SERVICE ACTIVITY [Hentrich et al. 2008]. The service notifies the process engine about terminating an activity when it receives a corresponding *TerminateEvent* from the conductor. Read and write operations on the container are synchronized using locking and unlocking mechanisms.

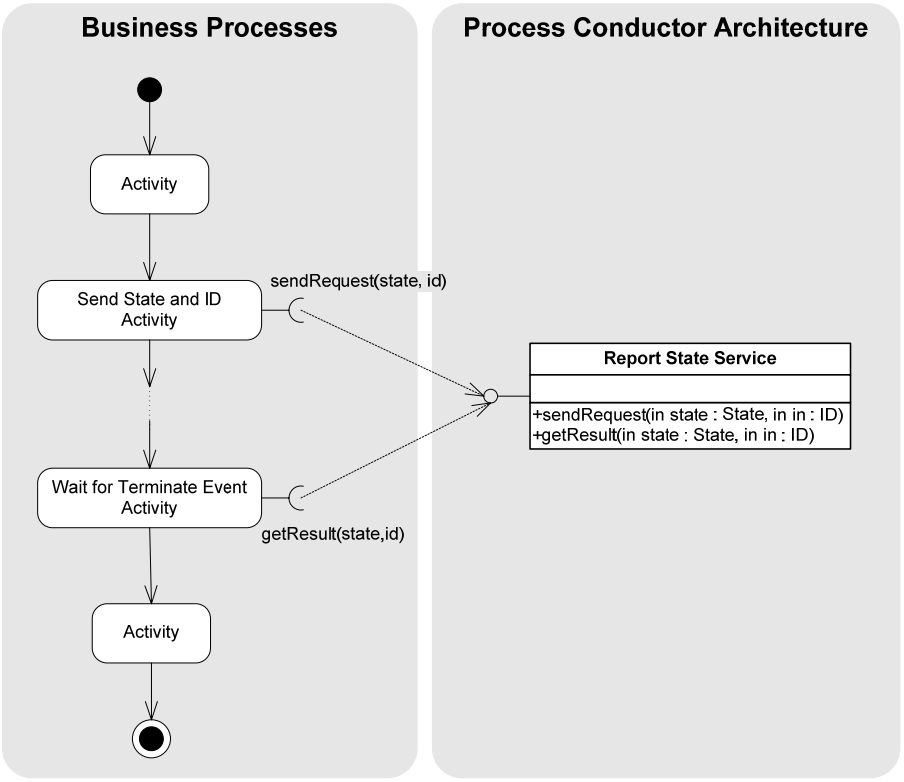
A variation of the pattern is that the registration of the state and process ID and the waiting position for the actual termination event to occur can also be designed as an ASYNCHRONOUS RESULT SERVICE [Hentrich et al. 2008].]. To model this variant, the *Report State Service* offers two methods: one that places the request and one that gets the result. The combination of state and ID may serve as a CORRELATION IDENTIFIER [Hohpe et al. 2003] for the second method invocation. The architectural concept then needs to change slightly according to the description of the ASYNCHRONOUS RESULT SERVICE pattern. Using this design principle the terminate event is rather captured by a pull mechanism from the perspective of the process conductor. The pull mechanism is represented by the second service invocation from the business process that actively asks for a result and the termination event might thus not be immediately reported to the process engine, i.e. in case the second method is invoked after the event has actually occurred.

On the contrary the original solution in the figure above seems rather to use a push-mechanism while following the EVENT-BASED ACTIVITY pattern, as the event is fired and reported to the process engine as soon as it occurs. However, from the viewpoint of the business process both scenarios follow a pull-mechanism, as all services are actively invoked and represent blocking calls. In the second scenario it is just two method invocations instead of just one. The second method to get the result represents the EVENT-BASED ACTIVITY in this second scenario.

The second variation of the solution can be used when sending the request needs to be decoupled from capturing the termination event. For instance, in case other process steps can be undertaken in the meantime but the conductor needs to be informed early. Doing it that way creates more time for the conductor to calculate the order of the terminate events, e.g. in case complex time consuming rules need to be applied and/or it is not necessary to report the termination event to the process engine as soon as possible.



The following figure illustrates the variation of the pattern using ASYNCHRONOUS REPLY SERVICE.



Example: Just-in-time Production Optimization

In a just-in-time (JIT) production scenario in the automotive industry the order of a car needs to be processed. The order arrives with given ordering details of the car model that needs to be manufactured in a JIT process. The parts for the car are delivered by different suppliers and the order details related to the parts delivered by a certain supplier are forwarded to each of the suppliers via a service interface. The manufacturing process in terms of the internal ordering, delivery, and assembly of the parts from those different suppliers needs to be coordinated. To coordinate the processes a MACROFLOW ENGINE [Hentrich et al. 2007] is used. The selected tool to implement the MACROFLOW ENGINE is WebSphere MQ Workflow.

The processes for ordering, delivering and assembling the parts need to be coordinated, as a parallel process instance is created for each supplier. Appropriate coordination of the process is crucial to optimize the manufacturing costs, e.g. reducing stock costs by agreeing service levels with suppliers in terms of delivery times and to place the order to the suppliers at the right point in time. In order to allow coordination of the processes and to allow optimization the PROCESS CONDUCTOR pattern has been applied. The timely coordination of orders to suppliers, parts delivery and assembly can thus be implemented using flexible orchestration rules. The rules can be modified according to improved service level agreements and to optimize the overall manufacturing process over time. The rules have been implemented and flexibly configured using the ILOG JRules [ILOG 2008] rules engine. The rules have been accessed by the PROCESS CONDUCTOR via a Java interface.

Known Uses

- The pattern has been used in projects in the telecoms industry to control technical activation of dependent products in larger orders. Each product can be processed in parallel to improve efficiency up to a certain point. Further technical activation of the products is then controlled by product rules, as certain products are dependent on each other, e.g. an internet account can only be activated if the physical DSL connection has been established.
- The pattern has been used to design synchronization points in parallel claims handling processes in the insurance industry. That is, a set of parallel sub-claims processes that belong to an overall claim is triggered off and can only move to a certain point. At this point the parallel processes need to be synchronized, i.e. the first process that reaches the point must wait until all others have reached their synchronization point too. What processes need to be synchronized is defined by configurable rules.
- In logistic processes in the transportation industry the pattern has been used to flexibly coordinate the transportation of goods delivered by different suppliers and parties. Orchestration rules have been used to allow flexible packaging and to coordinate between different types of transportation, e.g. trucks, planes, ships, and trains. That way it is possible to rather easily configure modified types of packaging and transportation due to changed conditions or different transportation criteria, e.g. security, delivery speed, and costs, which apply to different types of goods.

Conclusion

In this paper we have presented three patterns for synchronization of parallel and independently running business processes in a process-driven and service-oriented architecture. These patterns focus on coordinating the parallel and principally independent business processes via architectural solutions allowing architects to model the flexible synchronisation of the processes. The patterns are part of an ongoing effort to mine and document a pattern language for process-driven and service-oriented architecture. Previous parts of this pattern language have been published in [Hentrich 2004, Hentrich et al. 2007, Köllmann et al. 2006, Hentrich et al. 2008].

Acknowledgments

The authors would like to thank our shepherd Stephan Lukosch and the EuroPLoP 2008 writers' workshop for their valuable comments.

References

[Barry 2003] D. K. Barry. Web Services and Service-oriented Architectures, Morgan Kaufmann Publishers, 2003.

[Buschmann et al. 2000] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal: Pattern-Oriented Software Architecture – A System of Patterns, John Wiley & Sons, 1996.

[Channabasavaiah 2003 et al.] K. Channabasavaiah, K. Holley, and E.M. Tuggle. Migrating to Service-oriented architecture – part 1, http://www-106.ibm.com/developerworks/ webservices/library/ws-migratesoa/, IBM developerWorks, 2003.

[Gamma et al. 1994] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.

[Hentrich 2004] C. Hentrich. Six patterns for process-driven architectures. In Proceedings of the 9th Conference on Pattern Languages of Programs (EuroPLoP 2004), 2004.

[Hentrich et al. 2007] C. Hentrich, U. Zdun. Patterns for Process-Oriented Integration in Service-Oriented Architectures, In Proceedings of the 11th Conference on Pattern Languages of Programs, (EuroPLoP 2006), 2006.

[Hentrich et al. 2008] C. Hentrich, U. Zdun. Patterns for Invoking Services from Business Processes. In Proceedings of European Conference on Pattern Languages of Programs (EuroPloP) 2007, Universiätsverlag Konstanz, 2008.

[Hohpe et al. 2003] G. Hohpe and B. Woolf. Enterprise Integration Patterns. Addison-Wesley, 2003.

[ILOG 2008] ILOG. JRules. http://www.ilog.com/products/jrules/index.cfm, 2008.

[Köllmann et al. 2007] T. Köllmann, C. Hentrich. Synchronization Patterns for Process-Driven and Service-Oriented Architectures, In Proceedings of the 11th Conference on Pattern Languages of Programs, (EuroPLoP 2006), 2007.

[Schmidt et al. 2000] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. Patterns for Concurrent and Distributed Objects. Pattern-Oriented Software Architecture. J.Wiley and Sons Ltd., 2000.

[Völter et al. 2004] M. Voelter, M. Kircher, and U. Zdun. Remoting Patterns - Foundations of Enterprise, Internet, and Realtime Distributed Object Middleware, Wiley Series in Software Design Patterns, J. Wiley & Sons, October, 2004.

Appendix: Overview of Referenced Related Patterns

There are several important related patterns referenced in this paper, which are described in other papers, as indicated by the corresponding references in the text. The following table gives an overview of thumbnails of these patterns in order to provide a brief introduction to them for the reader. For detailed descriptions of these patterns please refer to the referenced articles.

Pattern	*Problem*	*Solution*
ASYNCHRONOUS RESULT SERVICE [Hentrich et al. 2008]	A communication between a service and a process flow needs to be modelled that is not a synchronous communication, but rather just places the service request and picks up the service result later on in the process flow, analogous to the well-known callback principle.	Split the request for service execution and the request for the corresponding result in two SYNCHRONOUS SERVICE ACTIVITIES and relate the two activities by a CORRELATION IDENTIFIER [Hohpe et al. 2003] that is kept in a control data object.
BUSINESS OBJECT REFERENCE [Hentrich 2004]	How can management of business objects be achieved in a business process, as far as concurrent access and changes to these business objects is concerned?	Only store references to business objects in the process control data structure and keep the actual business objects in an external container.
COMPONENT CONFIGURATOR [Schmidt et al. 2000]	How to allow an application to link and unlink its component implementations at runtime without having to modify, recompile, or relink the application statically?	Use COMPONENT CONFIGURATORS as central components for reifying the runtime dependencies of configurable components. These configurable components offer an interface to change their configuration at runtime.

Pattern	*Problem*	*Solution*
CORRELATION IDENTIFIER [Hohpe et al. 2003]	How does a requestor that has received a response know to which original request the response is referring?	Each response message should contain a CORRELATION IDENTIFIER, a unique identifier that indicates which request message this response is for.
EVENT BASED PROCESS INSTANCE [Hentrich 2004]	How can a process instance be automatically created in case an event based activity occurs in a business process that implies automatic process instantiation, e.g. a customer placing an order?	Externalise event based activities to an external event handler component and split the process model in several parts.
EVENT-BASED ACTIVITY [Köllmann et al. 2006]	How can events that occur outside the space of a process instance be handled in the process flow?	Model an event-based activity that waits for events to occur and that terminates if they do so.
FIRE AND FORGET SERVICE ACTIVITY [Hentrich et al. 2008]	A communication between a service and a process flow needs to be modelled that is not a synchronous communication, but rather just placing the service request without waiting for any result to be returned from the service.	Model a FIRE AND FORGET SERVICE ACTIVITY that decouples the request for execution of a service from the actual execution of the service.
MACROFLOW ENGINE [Hentrich et al. 2007]	How is it possible to flexibly configure macroflows in a dynamic environment where business process changes are regular practice, in order to reduce implementation time and effort of these business process changes, as far as the related IT issues are concerned that are involved in these changes?	Delegate the macroflow aspects of the business process definition and execution to a dedicated MACROFLOW ENGINE that allows developers to configure business processes by flexibly orchestrating execution of macroflow activities and the related business functions.
MACRO-MICROFLOW [Hentrich et al. 2007]	How is it possible to conceptually structure process models in a way that makes clear which parts will be depicted on a process engine as long running business process flows and which parts of the process will be depicted inside of higher-level business activities as rather short running technical flows?	Structure a process model into macroflow and microflow.
SYNCHRONOUS SERVICE ACTIVITY [Hentrich et al. 2008]	A synchronous communication between a service and a process flow needs to be modelled such that the process is able to consider the functional interface of the service and may react on the possible results of the service.	Model a SYNCHRONOUS SERVICE ACTIVITY that depicts the functional input parameters of the associated service in its input data objects and the functional output parameters of the service in its output data objects.

Developing GUI Applications: Architectural Patterns Revisited

A Survey on MVC, HMVC, and PAC Patterns

Alexandros Karagkasidis

karagkasidis@gmail.com

Abstract. Developing large and complex GUI applications is a rather difficult task. Developers have to address various common software engineering problems and GUI-specific issues. To help the developers, a number of patterns have been proposed by the software community. At the architecture and higher design level, the Model-View-Controller (with its variants) and the Presentation-Abstraction-Control are two well-known patterns, which specify the structure of a GUI application. However, when applying these patterns in practice, problems arise, which are mostly addressed to an insufficient extent in the existing literature (if at all). So, the developers have to find their own solutions.

In this paper, we revisit the Model-View-Controller, Hierarchical Model-View-Controller, and Presentation-Abstraction-Control patterns. We first set up a general context in which these patterns are applied. We then identify four typical problems that usually arise when developing GUI applications and discuss how they can be addressed by each of the patterns, based on our own experience and investigation of the available literature.

We hope that this paper will help GUI developers to address the identified issues, when applying these patterns.

1 Introduction

Developing a large and complex graphical user interface (GUI) application for displaying, working with, and managing complex business data and processes is a rather difficult task. A common problem is tackling *system complexity*. For instance, one could really get lost in large number of visual components comprising the GUI, not to mention the need to handle user input or track all the relationships between the GUI and the business logic components. Without proper system design, GUI code and business logic may get deeply intertwined with each other, which leads to another important issue: The *code organization*. A general requirement is that the source code be understandable, maintainable, and reusable. Clear code organization and coding principles are essential as regards communication among the developers within the whole system lifecycle.

Given these challenges, a (now) common approach in software engineering is to start with defining the system architecture. Iterative refinement leads then to the low-level design, where the detailed system structure is obtained. *Separation of concerns* is a basic principle underlying this process.

With time, a number of patterns for GUI applications have been proposed by the software community. The well-known Model-View-Controller (MVC) with its variants and, to a much lesser extent, the Presentation-Abstraction-Control (PAC) patterns provide a good starting point for the developers. However, when applying these patterns in practice, a number of problems arise both at the design and the implementation levels. For instance, an essential constraint is the GUI platform (or toolkit) used. It often assumes a specific way of how GUI applications are, or should be, developed (a *path of least resistance*), which may not comply with the pattern used. This should be taken into account when applying a particular pattern.

In large and complex GUI applications other issues become apparent as well, such as how the application's GUI part is constructed, how user input is handled, or how user-system dialog (interaction) is managed. Most of these problems are addressed to an insufficient extent (if at all) in the existing literature.

In this paper we revisit three well-known patterns for GUI applications – MVC, HMVC (Hierarchical MVC) and PAC – by identifying four typical problems in the development of GUI applications and discussing how they are, or could be, addressed by these patterns.

The paper is organized as follows. In the Background section we give a brief description of the MVC, HMVC, and PAC patterns, as well as present the four problems, which are then discussed one by one in the subsequent sections. The discussion is illustrated with numerous examples. Some implementation specifics for Java programming language and Java Swing toolkit are provided as well. We conclude our paper with a brief summary. Throughout the paper, we assume that the object-oriented approach is used.

2 Background

Patterns are usually applied within a systematic approach to system development, where the system architecture is first defined given the system requirements. Through the system design, following the separation of concerns principle, it is then refined into a set of classes, representing the detailed system structure. Each class is assigned specific responsibilities. At runtime, the system can be seen as a network of interacting objects.

Let's consider a typical usage scenario of a GUI application: A user wants to accomplish some (business) task. He/she selects the corresponding menu item, and an input dialog is displayed. The user types some data in and submits the dialog. The data is processed by the application, and results are displayed to the user.

Implementing such a scenario, we get a number of interacting objects with different functions. For instance, presentation (or GUI) objects comprise the GUI, display data to the user, and process user input. Other objects could be responsible for transforming between GUI- and application-specific data types. The input data is passed to the application logic objects and processed there. The results are then returned to the GUI part, and the visual state of the presentation objects is updated.

In a large application, with dozens of usage scenarios, the system design may contain hundreds of classes with intensive interaction among objects at runtime. The more features a GUI application provides, the tighter the relationships among objects are.

This makes the developing of GUI applications a rather difficult task. Typical questions would be:

- How to identify classes? How to assign specific functionalities to each class and design their interaction (as this can be done in different ways)?
- How to map classes onto implementation-level constructs (such as GUI toolkit widgets or Java classes)?
- How to initialize the resulting object network at runtime, instantiating particular objects and establishing relationships among them, i.e., initializing object references (known also as the *visibility problem* [Marinilli06])?

Apparently, we do not have to start from scratch when tackling these issues, since these seem to be in the realm of the patterns for GUI applications, such as Model-View-Controller or Presentation-Abstraction-Control (and their variations).

Here, we briefly review the patterns that are discussed in this paper. A detailed description could be found, e.g., in [POSA I] (see also the References section).

2.1 Patterns for GUI Applications

Model-View-Controller

Model-View-Controller (MVC) is probably the most popular and frequently cited pattern stemming from the Smalltalk environment ([POSA I], [KP88], [Burbeck92]). In MVC, a GUI application is built from components (objects) of three types: models, views, and controllers. The model represents the application logic – functions and data. The view displays the model data to the user (there can be several views connected to the same model component). The controller handles user input and forwards it to the model by calling the corresponding function(s). The model processes input and changes the application state. Such changes need to be communicated to the views depending on the model, as well as to the controllers, since the way they handle user input may also depend on the system state (e.g., enabling/disabling buttons or menu items in certain states). For this purpose, the Observer pattern is used. The views and controllers register themselves with the model. Upon state changes, the model notifies all the registered components, which then retrieve the required data from the model.

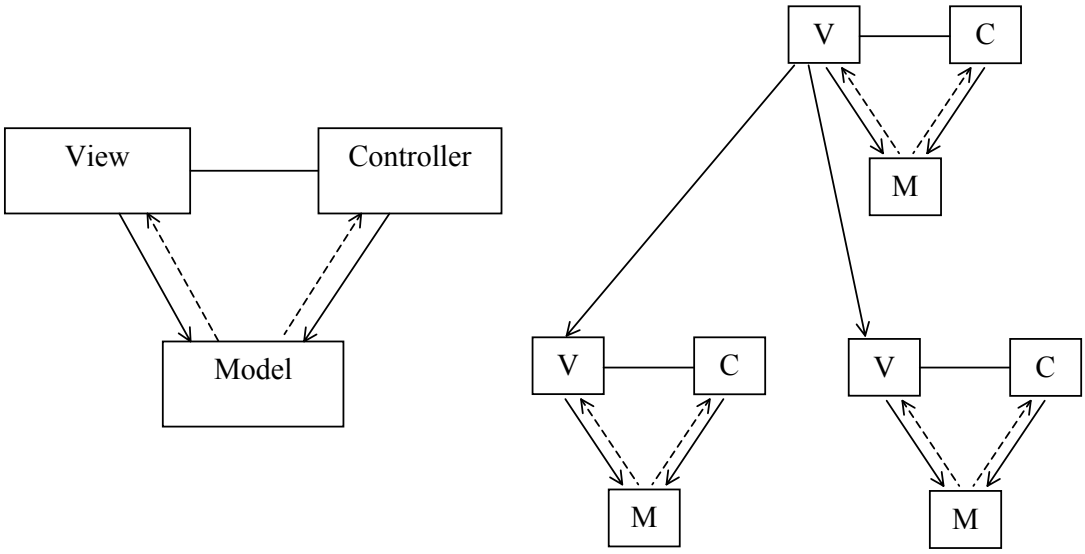


Figure 1. MVC and a hierarchy of MVC-triads

This is basic MVC. The whole application is then built as a hierarchy of MVC-triads organized around the view components, i.e., this hierarchy reflects the hierarchy of visual components (such as main window, menu bar, menus, panels, widgets) within GUI [Burbeck92].

Hierarchical Model-View-Controller

Hierarchical Model-View-Controller (HMVC) is an extension of basic MVC [CKP00]. A GUI application is modeled as a hierarchy of MVC-triads connected through the controllers.

There are some essential differences as compared to MVC. First, user input is handled now in the view component, which forwards it to its controller. The controller, in its turn, transforms user events into method calls on the model. Upon a state change, the model updates the view supplying it with the new state to be displayed (recall that in MVC the model only notifies its views upon state change, and the views retrieve then the data).

Another role of the controller is to enable communication with other MVC-triads. Such a communication is achieved through a hierarchy of controllers, thus providing a kind of the dialog control logic.

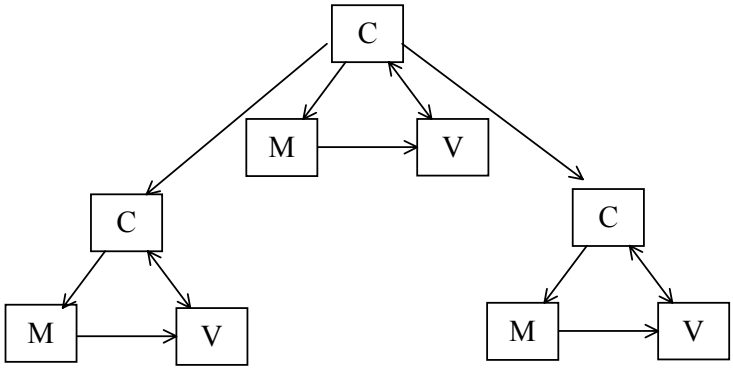


Figure 2. HMVC

Presentation-Abstraction-Control

Presentation-Abstraction-Control (PAC) organizes a GUI application as a hierarchy of cooperating *agents* ([Coutaz87], [POSA I]).

An agent (usually, it is a bottom-level agent) represents a '*self-contained semantic concept*' [POSA I] and is comprised of three components responsible for various aspects of the agent (that is, of the corresponding semantic concept). The presentation component is responsible for interaction with the user: It displays data and handles user input. The abstraction component represents agent-specific data. The control component connects the agent's presentation and abstraction parts with each other and is responsible for communication with other agents. When an agent wants to send an event to another agent, it forwards it its parent (intermediate-level) agent. The parent agent either sends the event to one of its other children, or it forwards the event to its parent, if it does not know what to do with it, and so on.

Intermediate-level agents serve two purposes. First, it is composition of lower-level agents. This is needed when we have a complex semantic concept represented by other concepts, which are then implemented as lower-level agents. Second, it is communication among agents, as described above.

The top-level agent implements the business logic and is responsible for the application-level GUI elements, such as menus or tool bar. It also coordinates all other agents.

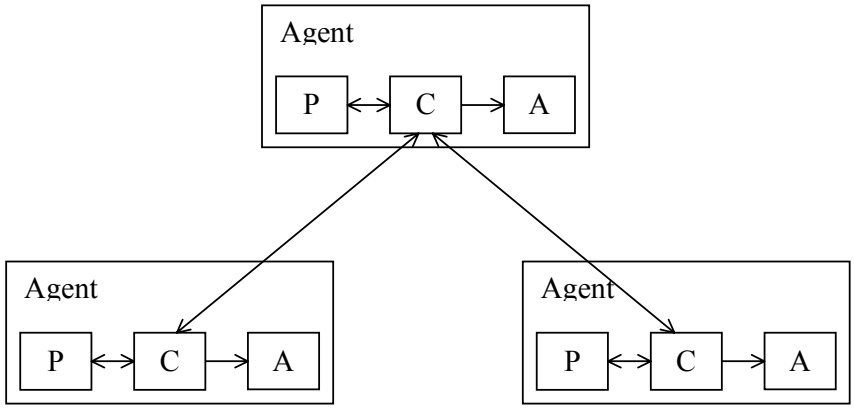


Figure 3. PAC

2.2 Typical Problems in GUI Development

The patterns presented above are similar to each other in that they all identify basic application components and assign certain functionalities to these components, such as displaying data to the user, handling user input, and processing the data in the application logic. These are, however, only some of common functionalities inherent to each GUI application. A number of other typical problems usually arise both at the design and implementation level, which becomes especially apparent when developing large and complex GUI applications.

In this paper we consider the following four of these problems:

- **Creating and assembling GUI (Content)**: The application's GUI part is comprised of visual components, or widgets, which are usually organized in a hierarchical manner into panels, menus and/or other containers within the main window or dialogs. Main problems here are how widgets are instantiated (in particular, regarding their relationships with other objects) and how they are assembled into the whole GUI (this is especially inherent to large GUIs).

- **Handling user input**: The GUI provides various input means to the user. Primary user input handling is usually implemented through callback methods, which are called by the GUI platform when the corresponding event occurs. In large and complex GUIs, with dozens of widgets, the code that handles user input may become rather large. Yet user events must be forwarded to the business logic for application-specific processing, which implies additional design-level issues.
- **Dialog control**: Dialog control is about managing user-system interaction scenarios, which may be rather complex involving several interaction steps. Roughly speaking, at each step the user provides some input (through the GUI), which is processed in the business logic, and the results are displayed (again, through the GUI). From the design viewpoint, this involves interaction among a number of objects from GUI and business logic, and the problem is to design and control such interactions.
- **Integrating GUI and Business Logic**: Eventually, the GUI must be coupled with the business logic. The way the business logic is designed is important with respect to how this coupling could be implemented.

The first two problems presented above are rather implementation-level ones. In this regard, an essential constraint is the GUI platform (toolkit) used, which assumes a specific way of how the application's GUI part should be developed (*a path of least resistance*). This should be taken into account when applying a particular pattern. The last two problems are more or less independent from the GUI platform.

The available literature on the patterns for GUI applications does not provide enough information (if any) on these problems leaving many questions open. So, the developers have to find their own solutions.

In the second part of the paper, we give a detailed consideration of these problems and discuss how they are (or could be) addressed by the MVC, HMVC, and PAC patterns, respectively. This work is based on our own experience and on the analysis of the existing literature.

2.3 The Smart UI Approach

The pattern-based approach to system development is not the only way the applications can be developed. Here we present the Smart UI approach, which will be used in the subsequent sections to illustrate the GUI development problems we discuss in this paper.

Consider a calculator GUI application having about 20-25 buttons, a text field, a few menus, and some other widgets. We can implement it just as one single class, which contains the GUI and user event handling logic, as well as mathematical operations.

In such simple applications developers can easily go without paying much attention to the system architecture and design. The point here is that there is no need to complicate things unnecessarily. Such a one-class solution is a particular case of what is known as the Smart UI approach [Evans03].

The main characteristic of the Smart UI, in general, is mixing business and presentation logic (like GUI creation and event handling). As Evans points out, “this happens, because it is the easiest way to makes things work, in the short run” [Evans03].

The Smart UI is a straightforward way to implement GUIs, which is very simple to understand. It is widely applied by developers, especially when working with popular visual programming tools (GUI Builders), which let the developers design the GUI with a kind of visual editor and automatically generate the source code, including empty callback methods to handle user input. The developers often directly implement the required business logic in these callback methods. Such a programming style can also be found in many books and articles on GUI programming.

However, as an application gets more complex, a solution based on the Smart UI approach degrades quickly – in the absence of any clear system architecture and design, with intertwined business and GUI logic, the system and the code become difficult to understand, maintain, reuse, or modify (turning the Smart UI into the Smart UI ‘anti-pattern’ [Evans03]).

3 Creating and Assembling GUI (Content)

A GUI typically consists of a main window, and a number of dependent secondary windows (basically, for output purposes) and/or dialogs (for user input). Windows and dialogs are composed of various visual components (widgets) that are organized into visual containers in a hierarchical manner. Hence, a window or a dialog can be seen as *a hierarchy of visual components*. A typical main window would contain the following areas ([CKP00], [Marinilli06]):

- Main working area (e.g., a drawing pane)
- Navigation (or Selection) area (e.g., a tree-based browser)
- Menu bar
- Tool bar
- Status line

As it could be seen, GUI may have a rather complex hierarchical structure. In this respect, we identify the following problems, which are especially inherent to large GUIs:

- How and where is each particular widget created (i.e., which object is responsible for creating and initializing a given widget)?
- How and where is the whole GUI assembled from particular widgets (i.e., which object (or objects) establishes relationships among widgets within the hierarchical GUI structure)?

From the implementation viewpoint, GUI platform (or toolkit) is the major constraint, since it provides widgets from which the GUI is constructed.

Regarding the GUI patterns, GUI toolkit implies another important issue. Each GUI pattern has elements (classes) that are responsible for the GUI (presentation) concerns. These elements have eventually to be implemented with the given GUI toolkit. As a consequence, this may require that we extend standard toolkit widgets to provide the pattern-related functionality. The widgets are then used in a specific way, which differs from the standard way intended by the toolkit, as expected by the developers (*path of least resistance*). This, in its turn, implies additional implementation effort.

This section is organized as follows. We start with the Smart UI approach to illustrate typical problems when creating and assembling the GUI. We then consider how MVC, HMVC and PAC address the identified problems.

3.1 Creating and Assembling GUI in the Smart UI Approach

Within the Smart UI approach, it is typical to implement the main window as a single class that instantiates all the widgets, initializes and composes them into visual containers, and eventually assembles the whole GUI. GUI builders – popular tools for GUI programming – encourage this way of creating GUIs, which is therefore often used in practice, especially by non-experienced GUI developers.

Let's see what the GUI creation code within the main window class includes. First, we would have a class instance variable (field) for each widget (both simple widgets and visual containers). Then, each widget must be instantiated and its properties must be set up. For instance, a menu item could be given a name, an accelerator key, a mnemonic key, an icon, and a tip text, and its state could be set to enabled or disabled. All this requires several lines of code. Listing 1 illustrates possible implementation in Java Swing.

Listing 1: Initializing a menu item in Java Swing

```
// JMenuItem saveMenuItem;
...
saveMenuItem = new JMenuItem("Save");
saveMenuItem.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_S,
                                                   ActionEvent.CTRL_MASK));
saveMenuItem.setMnemonic(KeyEvent.VK_S);
saveMenuItem.setIcon(new ImageIcon("resources/images/save.gif"));
saveMenuItem.setToolTipText("Save");
saveMenuItem.setEnabled(false);
```

At last, the main window class has to establish all the relationships among widgets within the visual component hierarchy, i.e., *assemble* the GUI. For instance, menu items must be added to their menus, which themselves must then be added to the menu bar.

Imagine now how the main window class would look like just with a hundred of menu items. It gets quickly very large, even if we take into account just the GUI creation and assembly code (without event handling or business logic, which the main window class would most likely contain in the Smart UI approach). And as it seems, the menu bar with all its menus and the tool bar (to a lesser degree) are the main contributors in this respect.

The code size and code organization problems can be solved by a number of well-known refactoring techniques ([Fowler99], [Marinilli06]). With the Extract Method refactoring technique, we can organize the code within the main window class, so that, e.g., the menu bar and all its content are completely created within a separate method. If this method becomes large, too, we can extract methods from it, which create particular menus within the menu bar, as it is illustrated in the Listing 2.

Listing 2: Extract Method

```
public class MyMainWindow extends JFrame {
   ...
   private void createMenuBar(){
      ...
      this.createFileMenu();
      ...// create other menus
    }

   private void createFileMenu(){
      // code that creates the File Menu
    }
   ... // methods creating other menus
}
```

The code size problem can be solved then by applying the Extract Class [Fowler99] or Extract Panel [Marinilli06] refactorings. With this technique, we extract all the GUI code for, say, menu bar into a separate class. The same can be done for tool bar, a specific panel or other containers within the main window.

3.2 Creating and Assembling GUI in MVC and HMVC

In MVC, GUI (presentation) concerns are handled by the view component. Logically, it is responsible for presenting application (business) data to the user. Eventually, each view has to be implemented with the given toolkit. Several designs are possible here.

Widget-level MVC

In the widget-level MVC approach, a view is a visual component (widget) from the toolkit. More precisely, each toolkit widget in the GUI is considered as a view component (which results in an MVC-based framework [POSA I]). Each widget has its model and controller components. Together, they form a simple MVC triad. The application is then created by instantiating such MVC triads for each GUI widget and assembling them together.

This approach originates from the Smalltalk environment, which provides predefined view, model, and controller classes for standard GUI widgets, like buttons, menu items, text fields, etc. ([KP88], [Burbeck92]). Java Swing is also an MVC-based toolkit: Each widget has its standard model and controller classes.

Widget-level MVC seems to work rather well, if an MVC-based toolkit is used and the application is simple. Let's consider potential problems that might arise when developing large and complex GUIs.

First, as in the Smart UI approach, it is the large main window class problem, if the whole GUI creation and assembly work is done in one class. Again, a solution would be to apply the Extract Method and Extract Class refactorings (see also the discussion in the previous section). For instance, with the Extract Class refactoring, we implement certain view components in separate classes. Each such view class instantiates all its child MVC triads and assembles its GUI part from the child view components.

Another problem concerns the model component associated with each widget. In MVC, the model is responsible for business logic and uses its own data type(s) to represent the business data to be displayed to the user. On the other hand, the view component uses, in general, another (usually toolkit-specific) data type(s) for the data it displays. So, we need to provide data type transformation. Let's consider this on the Java Swing example.

In Java Swing, each widget has its standard Swing model class, which maintains data displayed by the widget. However, this data is kept in Swing-specific format, which implies the following design problem: What if you want to implement your business data using other data types? In that case, the standard model class provided by Swing cannot be used directly. For instance, the `JTree` component requires a class that implements the Swing-specific

`TreeModel` interface as the associated model, which represents a hierarchical structure. And you'd most likely not want to model your business objects and relationships between them (which you want to display with the `JTree`) by implementing the `TreeModel` interface, but rather choose your own data structures.

So, how should we then connect a widget to the corresponding business data?

Extend Widget
One way is to extend the widget class associating it with our own model class that represents business data in its internal toolkit-independent format. The standard model class provided by Swing is no needed then. The extended widget retrieves the data to be displayed from the model and transforms it into its own format. The model has still to implement change notification mechanism (through the Observer pattern).

Extend Model
Another alternative would be to extend the Swing model class to connect it to our specific business object class. In this case, the extended model class can implement the required data type transformation as well.

As we can see, in both alternatives the developers have to extend the standard Java Swing functionality.

Consider now the situation when the toolkit does not support MVC, and we are going to apply MVC at the widget level, as described above. In this case, we would have to extend all the toolkit widgets we use to provide the MVC-specific functionality, such as connection to the controller and model components (see also the MVC liabilities section in [POSA I]).

This has some important implications for large and complex GUIs. First, for each widget in the GUI, we would have a separate class extended from the corresponding toolkit widget class. As a result, we would get a large number of new classes in our application (even for simple widgets, such as buttons or menu items), which need to be maintained. Second, the developers have to use the extended widgets in the MVC-specific way, which may differ from that the developers are familiar with (as suggested by the toolkit – path of least resistance). This means for the developers a new way of thinking about the widgets. The main window assembly problem is present here as well.

Concluding, we can say that for both cases we face the problem of being forced to extend standard toolkit functionality through implementing new classes in order to provide 'true' MVC nature of our application (especially for the toolkits that do not support MVC). Another problem is that applying MVC for each widget is a rather low-level and fine-grained approach with too many pattern-specific details (imagine having a separate MVC triad for each menu item!). This makes things unnecessarily complex and may discourage the developers, especially if a large GUI application is developed (see also the MVC liabilities section in [POSA I]).

Container-level MVC. HMVC Approach

This approach tries to avoid the problems of widget-level MVC we have discussed above. The key idea here is to move from the level of particular widgets to higher levels in the visual component hierarchy representing the application's main window (or any other window). Namely, MVC is applied at the level of visual containers – parts within GUI that group together other (usually related) visual components. Typical containers are panels, dialogs, menu bar and its menus, and tool bar, which are considered now as MVC views.

Each container-level view is implemented as a separate class through extending the corresponding container widget from the toolkit (recall again the Extract Class refactoring technique!) and adding the required functionality. In particular, it is responsible for handling its content, such as creating, initializing, and assembling together the visual components it contains. The key point here is that all the GUI-specific work for all the child widgets within a container-level view is done in standard toolkit way (Listing 3). We have no more to treat each child widget in the MVC way and create (a large number of) new classes for this purpose. Thus we avoid all the low-level details and complexity of the widget-level MVC. This is particularly important for toolkits that do not support MVC. And the developers can now work with the toolkit as they used to. Only the container classes must be adapted to MVC.

Listing 3: Container-level MVC

```
public class MyXYZPanelView extends JPanel {
   // Child widgets are used in standard Java Swing way
   private JTextField  myTextField;
   private JTextField  myAnotherTextField;
   private JLabel      myLabel;
   private JLabel      myAnotherLabel;
   private JButton     myButton;
   ...
   private JCheckBox   myCheckBox;
   ...
   public MyXYZPanelView (){
      ...
      this.myTextField = new JTextField(10);
      this.myAnotherTextField = new JTextField(10);
      this.myLabel = new JLabel("My Label");
      this.myAnotherLabel = new JLabel("My Another Label");
      this.myButton = new JButton("My Button");
      ...
      this.myCheckBox = new JCheckBox();
      ...
      // other initialization code
    }
   ...
   // other code
}
```

This approach is used in HMVC [CKP00], where main GUI parts, such as menu bar, navigation pane, main content pane (working area), and status pane, are modeled as views. These views are assembled then within the main window, which is the root view component.

The main design issue with the container-level MVC approach is the granularity level, i.e., which containers should be modeled as MVC views. In the hierarchy of visual components representing the structure of the main window there could be several layers of intermediate containers, depending on the GUI complexity. For instance, the menu bar (container) is usually comprised of several menus (containers), where each menu consists of either menu items (leaf nodes) or other submenus (containers), and so on. Complex forms are often composed from a number of panes (e.g., tab panes), where each pane may have its own sub-panes.

Another aspect is that sometimes we have to extend the functionality of particular widgets to provide some specific behaviour not supported by the toolkit used. Naturally, we would have a separate class for such a widget. We can then regard it as a separate view component and provide the corresponding model and controller components. Alternatively, we can treat it as a new "standard" widget just as other child widgets within a container.

So, we can put a more general question: Which visual components within GUI should be considered as MVC views?

The following two rules-of-thumb could be suggested in this regard:

- If a child container within a given container has complex structure (contains many widgets), then implement this child container as a separate view.
- (Optional) If a widget (visual component) within a container must provide some specific functionality, then implement this widget as a separate view.

The key point here is to find a balance between the one-class solution and applying MVC for each particular widget. With this, we are trying to avoid the two extremes: having a single monster class that implements everything or having too many classed with low-level details and unnecessarily complex design.

The following examples illustrate the idea. Consider first a menu bar. If is has two or three menus, each consisting of a couple of menu items, then we could model the whole menu bar as a single view component. If, however, some menu has, say, more than 5-7 menu items, it would be a candidate for a separate view. The same applies then to each particular menu. Until a menu remains simple, it is modeled as one view. When it becomes more complex, with large submenus added, then it is time to think about extracting submenus into separate views.

Consider now a navigation pane with two navigation trees. It could be implemented as a tabbed pane, with two tabs each containing a tree component. Though we have few widgets here, we would rather have a separate view for each tree, because each implements a specific behaviour and we must customize the tree widget from the toolkit. Another view would be the navigation pane itself.

As a counter example, consider a pane comprised of, say, 20 child panes, each having few (e.g., 3-5) simple widgets like buttons and text fields. For instance, this could be a pane representing various parameters of some physical process, with a couple of controls to regulate each parameter. So, what to do in this case? The whole pane having dozens of widgets is rather large to put all its content into just one view (i.e., class), but each of its child panes is rather simple to be considered as a separate view.

3.3 Creating and Assembling GUI in PAC

The way the GUI creation and assembly problem is addressed in PAC is logically quite different from, and more complicated as, that of MVC. So, we give first some details from [POSA I] regarding the presentation in PAC before we discuss the implementation issues.

First, an application is organized as a hierarchy of agents, which represent 'self-contained semantic concepts' (see also PAC description in Section 2.2). Each agent has the presentation component, which handles agent's GUI aspects (input and output) and interacts with the agent's control component. What the presentation component actually is depends on the agent's type. The presentation of the top-level agent '*includes those parts of the user interface that cannot be assigned to particular subtasks, such as menu bars…*' [POSA I]. The presentation of the bottom-level agents '*presents a specific view of the corresponding semantic concept, and provides access to all the functions users can apply to it*' [POSA I]. There are also intermediate-level agents. Some of them represent complex abstractions (concepts) from the business domain, which are comprised of other abstractions. In this case, the presentation of an intermediate-level agent handles the visual aspects of the corresponding compound abstraction.

Well, a little bit complicated as compared to MVC… An important aspect to note here is that the design of the whole application and its GUI part in particular is driven by agents. Regarding GUI issues, this means that we take, roughly speaking, a business object from the business model and then devise how it should be displayed to the user. This differs from MVC and HMVC where the view components represent widgets or containers within the main window, i.e., the application design is strongly influenced by the visual design of the GUI.

More insight on what is hidden behind the agent's presentation component could be deduced from the following statement: '… *All components that provide the user interface of an agent, such as graphical images presented to the user, presentation-specific data like screen coordinates, or menus, windows, and dialogs form the presentation part*' [POSA I].

This helps in reasoning about the implementation issues of the presentation component. First, for a simple agent, its presentation part could be implemented as a single class, which handles output, user input, and interaction with the agent's control component. Such a class could be derived from the toolkit widget class suitable to handle agent's visualization. For instance, for an agent representing some value label or text field widgets could be used. As it can be seen, this approach is very similar to applying MVC at the widget level. Therefore, if we have a large number of simple agents, we'll face similar problems. Namely, we have to extend standard toolkit widgets to provide PAC-specific functionality and we get unnecessarily complex design with too many low-level details. These issues are also discussed in the PAC liabilities section in [POSA I], and a good example is provided – a graphical editor where each graphical object is implemented as an agent.

A potential solution would be to choose the appropriate granularity level of agents to control their number. With this, the concepts from the business domain implemented by simple agents are handled within more complex agents. Note that this approach is similar to the container-level MVC and HMVC, where container-level views handle all their child widgets,

instead of having an MVC triad for each particular widget. The difference is that in MVC and HMVC we deal with the hierarchy of visual components, while in PAC it is the hierarchy of agents. Regarding the graphical editor example [POSA I], we would have an agent for the graphical editor only, but not for each graphical object. The editor agent then handles all the graphical objects internally.

In such complex agents, the presentation part gets also more complex. As mentioned above, it may include menus, window, dialogs, etc., which could be implemented as a set of (interacting) classes. For the interaction with the outside world through the agent's control component, the Façade pattern could be used [POSA I]. In this case, the Façade object hides the internals of the presentation part from the control component and is the only PAC-aware object of the agent's presentation. An essential consequence of this design solution is that it allows the developers to implement the visual components within the agent's presentation part in the toolkit-specific way, using standard toolkit widgets, which might need to be customized to meet some domain-specific requirements. But we do not have to implement any PAC-related functionality in the visual components (recall the containers-level MVC and HMVC!).

To illustrate these ideas, consider the graphical editor example with an agent representing the editor's drawing pane. Suppose that the agent's presentation part includes the drawing pane itself, various dialogs and popup menus, and a tool palette. Each of these visual components could be implemented in a separate class (probably with a number of additional helper classes). We would also have a Façade class. Another alternative would be to have only the drawing pane in the presentation, while the tool palette, dialogs, and popup menus are implemented as child agents of the drawing pane agent. Note that none of the visual components implements any really PAC-specific functionality.

The next problem may arise in the top-level agent. As stated above, its presentation part is responsible for the menu bar. Again, if menus comprising the menu bar have many items, this would result in a very large presentation component. A potential solution is to implement menus as separate agents, which is equivalent to the Extract Class refactoring technique. However, these new agents would imply additional interaction issues, which are discussed below (Section 5.3).

So far, we have considered how and where GUI parts are created in PAC. Now we need to assemble them. PAC does not address this issue explicitly (as given in [POSA I] or [Coutaz87]). In general, agents should be extended to provide functionality for GUI assembly. For instance, a parent agent may require its children for visual components from their presentation parts to construct its part of GUI. The major problem here is that the PAC agent hierarchy does not fully match the hierarchy of visual components. Consider again the drawing editor example. Suppose the tool palette is implemented as a child agent of the drawing pane agent, but is presented visually as a tool bar, which is a child visual component of the main window and should therefore be initialized by the top-level agent. For this purpose, the top-level agent must require the drawing pane agent for the tool palette. The drawing pane agent, in its turn, forwards the request to the tool palette agent, its child. All this just increases coupling among agents and their complexity.

4 Handling User Input

The user interacts with the system through the GUI using various input devices. Events from these devices are delivered by the operating to the application. In the application, user input events are usually handled in the callback methods. The details of how this is done are toolkit-specific.

After a callback method has intercepted a user event, it must be processed in an application-specific way. A typical way, which is often used in practice (Smart UI approach!), is to implement application-specific event processing logic in the callback methods. This works rather well in simple cases. However, application-specific event processing may be rather complex and involve components from business and/or presentation logic. So, we believe that a better design would be to just intercept the user events in the callback methods (we call this *primary event handling*) and forward them to other application components for application-specific processing.

In large GUIs, comprised of dozens of widgets and providing rich interaction features to the user, primary event handling code implemented in the callback methods may become rather large. A proper organization of this code is therefore an important task.

Let's look at how the patterns address this issue.

4.1 Handling User Input in MVC and HMVC

In MVC, user input is handled by the controller component, which translates user input into the method calls on the model or view components. As with the GUI creation and assembly problem, concrete implementation depends on at what level MVC is applied.

Consider first the widget level MVC approach. The controller component handles events on the corresponding widget (the view). We first create the widget. We then create its controller and associate the controller with the widget. From the implementation viewpoint, we have to provide event handling callback methods in the controller class and register these event handlers with the widget. This is rather straightforward to implement in many toolkits and platforms (like Java Swing and SWT, or .NET Windows Forms in C#, or Qt), if you write the code yourself. If you work with an IDE or a visual programming tool, you can get the event handling code (empty callback methods) generated automatically for you by the IDE. However, modifying this code to get pure MVC design would be a bad idea.

As for the GUI creation and assembly, we get similar problems with event handling. The main issue is that having a separate class for each controller results in a large number of small classes each providing probably only one or a couple of callback methods. This unnecessarily complicates the design. The main source of such controller classes would be menu items, tool bar buttons, and widgets of complex dialogs.

Therefore, we find it more reasonable to shift to the layer of visual containers, as it is done for the GUI creation and assembly problem (container-level MVC). There, certain visual

containers within the GUI are implemented as container-level views. For each such view, we have to define the controller component. Such a controller handles all the events that occur on the visual container itself, as well as on all the child widgets of the container. Thus, we avoid having a separate controller class for each particular widget.

This idea is also used in HMVC [CKP00]. The difference is that in HMVC user input handling is assigned to the view component (presumably stemming from the PAC pattern), while the controller component of an MVC-triad is responsible for communication with other MVC-triads. Thus, we get two slightly different approaches. Pure MVC separates presentation from user input handling, resulting in two different logical components (the view and the controller), while HMVC handles both issues in one component (the view).

As an illustration, let's consider how a container-level controller could be implemented in Java Swing. A common way to handle a user input event on a widget is to provide a class that implements the corresponding `Listener` interface, where event handling callback methods are declared. Such a listener class is then registered as an event handler with the widget by calling the corresponding `addXXXListener(XXXListener l)` method on that widget.

Suppose we are implementing event handling for a dialog with, say, a text field and a couple of buttons among other widgets.

Consider first pure MVC approach. Here, we have a separate controller. The first implementation alternative is when the controller class implements the required `Listener` interfaces. In our example these would be `ActionListener` and `KeyListener` (if we want to handle each particular keyboard input event on the text field), as shown in Listing 4.

Listing 4: Controller class implementing Listener interfaces

```
public class MyDialogController implements ActionListener,
                                            KeyListener {
  ...
  // ActionListener interface
  public void actionPerformed(ActionEvent ae){
     // event processing code
     // Note! We must identify here on which widget the event has occurred!
   }

  // KeyListener interface
  public void keyPressed(KeyEvent ke){
     // event processing code
   }

  public void keyReleased(KeyEvent ke){
     // event processing code
   }

  public void keyTyped(KeyEvent ke){
     // event processing code
   }
  ...
}
```

The next step is to register the controller as listener on all the widgets of the dialog class. Here, we have two options, depending on who performs the registering. An easier way is to pass an object of the controller class to the dialog (the view), which registers the controller as event listener on all its widgets (Listing 5).

Listing 5: Registering controller on widgets

```
public class MyDialog extends JDialog {
  ...
  JButton myButton;
  JButton myAnotherButton;
  JTextField myTextField;
  // Other widgets
  ...
  public void registerController(MyDialogController controller){
    ...
    this.myButton.addActionListener(controller);
    this.myAnotherButton.addActionListener(controller);
    this.myTextField.addActionListener(controller);
    this.myTextField.addKeyListener(controller);
    ... // and so on
  }
  ...
}
```

The second option is to perform the registration in the controller class, which implies the controller to have access to the dialog's widgets. As the widgets are class members in the dialog class, at least package access must be provided. If we want to keep them private, we should add the corresponding getter methods to the dialog class. So, the first alternative looks more elegant.

Note that the controller is registered as listener on several widgets, which may produce the same types of events (e.g., both buttons and text field in the example above fire `ActionEvents`). Hence, the controller must be able to find out in its callback methods on which widget a particular event has occurred. This could be done with the `getSource()` method called on the event passed as a parameter to the callback method. The `getSource()` method returns an instance of the `Object` class, which is then compared to each widget of interest to find out the event source. With this, we get conditional if-else-…-else blocks within callback methods, which may get rather large for complex views (imagine, for instance, such an if-then-else construct for a menu bar with all its content!). But it also means that the controller class must have access to the widgets of the dialog class!

The second alternative is to implement the MVC controller with Java anonymous inner classes. Using this feature of Java language, the controller does not need to implement `Listener` interfaces anymore, but needs to have access to the view's widgets. Listing 6 illustrates this approach (we assume that `myButton` and `myTextField` class members are defined with package access in the `MyDialog` class, see also Listing 5). Thus, we do not have anymore to work with cumbersome if-else-…-else constructs. All the event handling code for a particular widget is now localized within few (usually one) inner classes. If we

need to change something in that code, we don't have to run through all the if-else-…-else blocks looking for the code to be changed.

Listing 6: Implementing controller with anonymous inner classes

```
public class MyDialogController {
  MyDialog dialog;
  ...
  public void registerControllers(){
       this.dialog.myButton.addActionListener(
          new ActionListener(){
              public void actionPerformed(ActionEvent ae){
                    // event processing code
              }
           }
       );

      this.dialog.myAnotherButton.addActionListener(
          new ActionListener(){
              public void actionPerformed(ActionEvent ae){
                    // event processing code
              }
           }
       );

        this.dialog.myTextField.addActionListener(
          new ActionListener(){
              public void actionPerformed(ActionEvent ae){
                    // event processing code
              }
           }
       );

       this.dialog.myTextField.addKeyListener(
          new KeyListener(){
               public void keyPressed(KeyEvent ke){
                    // event processing code
                }

               public void keyReleased(KeyEvent ke){
                    // event processing code
                }

               public void keyTyped(KeyEvent ke){
                    // event processing code
                }
           }
       );
       ... // and so on
  }
  ...
}
```

We can also use 'normal' inner classes as event handlers for particular widgets inside the controller class. They would have to implement then the corresponding `Listener` interfaces.

Consider now the HMVC approach. In this case, we implement event handling in the view class. First, we avoid the visibility problem inherent to MVC where the controller is

implemented as a separate class. The two alternative solutions described above can be applied here as well. In the first, the view class must implement all the required `Listener` interfaces. This solution is used in [CKP00]. The view registers itself as a listener with each of its child widgets, which results again in if-else-…-else blocks within the callback methods to find out the source of an event.

The second alternative is to have anonymous inner classes within the view class that implement callback methods (see also Listing 6). We find this solution more elegant, as we do not have to do with the if-else-…else blocks.

In general, the view in HMVC is more complex than in MVC, since it is responsible for GUI creation and assembly, as well as for user input handling.

4.2 Handling User Input in PAC

In PAC, user input handling, along with the GUI concerns, is assigned to the agent's presentation component. As for the GUI creation and assembly problem, the details of how event handling could be implemented are left open. In general, it would depend on how the GUI is implemented, which has been discussed in Section 3.3. So, the implementation is completely up to the developers, and we just give some general ideas.

If agent's visualization is simple, we implement the agent's presentation component as a single class. Event handling is provided in this class through callback methods. In more complex cases, the presentation may include several visual components, like windows, dialogs, menus, etc., which are responsible for various visual aspects of an agent. Each complex visual component is implemented in a separate class, probably with some additional helper classes (see also Section 3.3). So, we need to provide event handling for these components. As PAC does not specify how this should be done, we can, for example, apply the solutions for MVC and HMVC discussed in the previous section. Namely, we can either put event handling logic (callback methods) into the visual component class itself (HMVC approach), or implement it in a separate class (MVC approach). Using Java Swing, this could be done either by implementing the corresponding `Listener` interfaces or by using anonymous inner class, as shown above. The callback methods process the events and send them to the agent's control component, or to the Façade object that could be used to hide the inner structure of the agent's presentation from the agent's control component.

4.3 Application-specific Processing of User Input

So far, we have discussed the design and implementation of the primary user input handling logic, which only intercepts user events in callback methods. Application-specific event processing logic is provided in other parts of the application (separation of concerns). Thus, we have to couple our callback methods with that logic. In MVC, the controller transforms user events into the calls on the model. In HMVC, the view forwards user input to the controller, which decides then what to do with it. The same is done in PAC. Details are discussed in the following sections.

5 Dialog Control

The problems discussed so far pertain to the presentation logic and its main responsibilities – user input and output. The application's presentation, or GUI, part has to be coupled with the business logic, where the user input is processed and the output is provided that is then displayed to the user. A number of issues may arise in this regard.

Consider a UML graphical editor. It would most likely have a tree-like navigation area representing UML diagrams and their elements, and a drawing pane to create UML diagrams. A typical usage scenario within such an editor would be to change some properties of a UML element. This can be done in the following manner: The user selects the required UML element in the navigation tree, gets a popup menu by clicking the right mouse button, and then selects the Properties menu item in the popup menu. Upon this action, the Properties dialog is displayed, where the user can edit various data related to the selected element. When the user clicks the OK button, the input is forwarded to the business logic and processed there. After the input has been processed, the GUI is updated to reflect the changes made on the selected element. In particular, these changes are visible both in the navigation tree and in the drawing pane.

Note how many things are done before user input is sent to the business logic: We need to display first the popup menu and then the Properties dialog, which must be populated with the data (state) of the selected UML element. After the input has been processed, we must communicate state changes to several parts within the GUI.

How could such an interaction be designed and implemented? And where should the corresponding logic reside, i.e., how do we assign responsibilities among our components? For instance, an interesting question in the above scenario is who (which object) creates and displays the popup menu and the Properties dialog?

The logic responsible for user-system interaction is often put into the event handling callback methods (the Smart UI approach). This works rather well in small applications with simple interaction scenarios (such as the calculator example from Section 2.3).

However, for complex GUIs we'll face certain difficulties. Consider again the UML editor example and implementation of the popup menu. We may create and display the Properties dialog in the callback method for the Properties menu item. The problem is that we must populate this dialog with the data of the selected UML element. As this data is obtained from the business logic, we have to access the business logic from the callback method. Another solution would be to provide the Properties dialog with a kind of identifier of the selected element and let the dialog obtain itself the required data from the business logic. The dialog also needs access to the business logic to send the user input when the OK button is pressed. After the input has been processed, the state changes to the navigation tree and the drawing pane. This could be done either by the dialog or by the business logic. In both cases, we get additional dependencies among different parts of the application, which need to be initialized and maintained (the visibility problem [Marinilli06]).

Another problem is to provide data type transformation, since in general, the data types used in the GUI toolkit widgets and the business logic are different.
To address these issues, the dialog control logic is provided, which controls and manages complex user-system interactions and performs data type transformations. The dialog control decouples, and glues together, the presentation and the business logic. In this section, we discuss how the patterns address the dialog control problem.

5.1 Dialog Control in MVC

In the MVC literature (e.g., in [POSA I]), simple (basic input-process-output) interaction scenarios are mainly considered, where user input is first handled by the controller, which transforms it into the method call(s) on the model. The model changes the system state and notifies all the dependent views and controllers upon that change (using the Observer pattern). The views retrieve the data from the model and display it to the user. Alternatively, the controller forwards user input directly to the view, if it is about changing the GUI visual state only and doesn't relate to the business logic (e.g., zooming).

The dialog control functionality is therefore distributed between the controller, model, and view components (in addition to their main responsibilities as defined in MVC):

- The controller transforms low-level toolkit-specific user input events into application-specific method calls.
- The model implements notification of the dependent view(s) and controllers, while its basic concern is business logic.
- The view retrieves data from the model and performs data type transformation, while its main responsibility is displaying data to the user.

This implies that separation of concerns is not completely achieved. As a result, the view is tightly coupled to the model and its data types [POSA I]. So, any change in the model implies changes in the dependent view(s).

Despite these shortcomings, such an interaction mechanism works rather well in simple cases, where interaction is confined within a model and its dependent view(s) and controller(s). However, MVC does not address more complex interactions, where several GUI and business logic elements are involved. As a result, the developers have to find their own solutions.

Let's take the above scenario with the Properties dialog in the UML graphical editor, which is rather complex. How could we implement it with MVC? First, consider what GUI parts we have in this scenario, which we would model as MVC-triads. Most likely, these are the tree-like navigation area MVC, the drawing pane MVC, the Properties dialog MVC, and the popup menu MVC (in the latter case, the corresponding model component could be the business object implementing the UML model element, which is represented by the selected element in the navigation area).

The scenario could be implemented as follows:

1. The controller of the navigation area MVC handles mouse events. Upon clicking the right mouse button, the controller should first identify the element in the navigation

area to which the right click relates, and then create and show the popup menu for this type of elements (there are usually several types of elements, each having specific popup menu or at least some specific menu items within the popup menu). The popup menu (the view) must be associated with the corresponding business object (the model). The controller for the popup menu is created and associated with the popup menu here as well (alternatively, the popup menu could create its controller itself).

2. When the user selects the Properties menu item, the event is handled in the controller of the popup menu. The controller's event handling callback method must create and display the Properties dialog (the view) and associate it with the corresponding controller and model, which is the same business object as for the popup menu.
3. As the Properties dialog is displayed, the user performs the required changes and presses the OK button. This event is handled in the controller of the Properties dialog MVC, which gathers user input and sends it to its associated business object (the model) by calling the corresponding method on it.
4. Upon this, the business object validates the user input and changes the object's state. The next step is to communicate this change to the views. These are the dialog itself, the drawing pane, and the navigation area. Moreover, if the model state has been saved prior to this state change, the File menu and the tool bar should be notified as well to enable the Save menu item and the Save button, respectively.

In order to implement the state change notification in Step 4, we need to register all the GUI components (views) being notified with the model component of the Properties dialog MVC. And this should be done in the controller (namely, in the corresponding callback method) of the popup menu MVC, which creates and initializes the MVC-triad representing the Properties dialog (Step 1). This implies that the popup menu controller should keep all these logical relationships, which is actually not its concern, and be also initialized with the references to these GUI components (the visibility problem!).

Another problem is that registering the drawing pane view on the model of the Properties dialog MVC would violate the MVC design, since we get the drawing pane view associated with two models – one is the drawing pane's own model and the other is the model of the Properties dialog. This raises a more general question of what the models in MVC actually are. In particular, how many model objects should we have in our application? Should we have a separate model component for each business object or just one model that hides all the business logic? This is an important design issue, which is discussed in section 6.1.

In general, the problems arise in scenarios where interaction involves several MVC-triads. In complex GUIs this leads to tight coupling among objects and to cumbersome and involved code, which is hard to write and understand, because it is very difficult to follow all the relationships among objects and provide reference initialization.

5.2 Dialog Control in HMVC

HMVC tries to address the dialog control issues inherent to MVC by providing a means for communication among different GUI parts. MVC-triads in HMVC are organized hierarchically according to the hierarchy of visual components, with the main window as the root (see also Section 3.2). The triads are connected through the controller components. The resulting controller hierarchy is responsible for the communication among different MVC-

triads, thus providing a solution to the dialog control problem. Presumably, this mechanism has been adapted from the PAC pattern and is discussed below.

5.3 Dialog Control in PAC

PAC explicitly addresses the dialog control problem. The application is modeled as a hierarchy of interacting agents. Within each agent, the control component is responsible for the agent's interaction with other agents. The control also mediates the agent's presentation and abstraction components. In particular, it performs data type transformation. Hence, each control component has two basic roles – it provides internal interaction between agent's parts and external interaction with other agents [POSA I].

Interaction between agent's components is rather simple. The presentation part forwards user input to the control and receives the data to be displayed (already in presentation-specific format) from it. The control obtains this data from the abstraction part, where it is kept in the application-specific (business logic) format.

For complex presentation parts, we have to provide interaction among visual components that comprise the agent's presentation. Recall the graphical editor example from Section 3.3. The presentation part of the drawing pane agent includes the drawing pane, the tool palette, maybe several dialogs and popup menus, which interact with each other. For instance, upon selecting a menu item in the popup menu, certain dialog has to be displayed, while when the user submits this dialog, the drawing pane might need to be updated. A straightforward way would be to implement the interaction functionality within the visual components themselves. However, this may result in a tight coupling among them, if the interaction is intensive. The Mediator pattern could be applied to decouple the visual components from each other. As an option, the mediation role could be assigned to the Façade object, which is used to hide the visual components from other components of the agent (as discussed earlier in Section 3.3). A potential problem here is that the Façade object may get rather complex. Basically, it is again the agents' granularity issue, where the developers have to find balance between the agent number and agent complexity.

Interaction among agents is more complex. In general, it is about exchanging command, data, and state change events. For example, upon selecting a menu item, the top-level agent, which is responsible for menus, must send the command message to the corresponding lower-level agent(s). Regarding data exchange, the entire application model in PAC is implemented in the abstraction component of the top-level agent [POSA I]. Abstraction components of other (basically, bottom-level) agents keep agent-specific data, which is part of the application model. Therefore, this data is obtained from the top-level agent. So, in general, the agent's local data must be kept synchronized with the application data in the top-level agent. As a consequence, when the user provides some input to a bottom-level agent, this input is first sent up to the top-level agent, which processes the input and changes the system state in its abstraction component. The state change is notified back to the bottom-level agent (and all other agents depending on this data). And only then retrieves the bottom-level agent the data from the top-level agent and updates its own abstraction and (most likely) presentation components.

[POSA I] specifies two basic mechanisms for inter-agent interaction. We give here brief descriptions thereof:

- **Composite Message Pattern**. Agents provide two methods to send and receive messages, containing the information about what is sent, like message type and content. For incoming messages, the developer must implement the logic that analyses the content of a message and decides what to do with it – process locally by sending it to the agent's presentation or abstraction components, or forward the message to another agent. From the implementation point of view, this may result in large and complex if-then-else blocks within the message processing methods. Another issue is identifying appropriate event types and introducing new ones.
- **Agent-specific Interfaces**. Each agent implements its specific interface used by other agents to interact with it. With this, we can avoid having complex if-then-else constructs inherent to the Composite Message pattern, but the agents become tightly coupled to, and dependent on, each other.

The drawbacks of both approaches are tolerable per se. However, they may become a real nightmare when applied in PAC. The reason is that PAC agents do not interact with each other directly, but through the agent hierarchy organized around the control components. For instance, the bottom-level agent in the data exchange scenario does not interact with the top-level agent, but rather with its parent agent, which, in its turn, forwards the request to its parent, and so on, until the request arrives the top-level agent. This means, however, that all the intermediate agents participate in this interaction and each of them must provide the corresponding logic, which is basically the request forwarding logic. For a Composite Message approach, we'll get large if-then-else decision-making block within the message receive method. When using the agent-specific interfaces approach, each intermediate agent must expose interfaces of all the agents below it (the child agents) to the agents above it (in particular, to the top-level agent), and vice versa.

We find both approaches completely unsuitable for large and complex GUIs. Again, the reason lies, to a greater extent, in the hierarchy-based agent interaction. It is a huge effort to implement such an interaction mechanism, which is also difficult to adapt and extend. For instance, if you have to extend the interaction logic between two agents, all the intermediate agents must be adapted to provide it, either through changing the message forwarding code in the if-then-else constructs, or through extending the agents' interfaces.

It is also not a very nice solution from the software design in general, and separation of concerns in particular, viewpoint. Indeed, why should we make the intermediate-level agents lying between two interacting agents in the agent hierarchy be aware of, and participate in, the interaction between these two agents? Why can't these agents interact directly with each other? One possible answer is that direct interaction among agents would break the tree-like agent hierarchy.

Given these problems, other approaches have been proposed. [POSA I] specifies also an agent interaction mechanism based on the Publisher-Subscriber [POSA I] or Observer [GoF] patterns. All the agents (subscribers, or observers) that depend on data or events of a specific agent (publisher, or subject) register themselves on that agent, which notifies them when changes occur. Upon a notification, the dependent agents update their states by retrieving the data from the publisher agent. Therefore, agents interact directly with each other. The Composite Message pattern or agent-specific interfaces approach could be applied here. The

only problem is the registration, as the interacting agents may reside in different parts of the hierarchy and we have to make them aware of each other.

[Wellhausen] provides a solution that solves this problem. It uses the Event Channel variant of the Publisher-Subscriber pattern, which decouples publishers and subscribers. Event channels are given as a system service. Publishers provide events of certain type for a specific channel. Subscribers register themselves with this event channel for events they want to receive. So, the subscribers must know only the event type.

[HO07] proposes a Hierarchical Service Locator mechanism. First, each agent (using PAC terms) implements the Service Locator interface and registers its interface for interaction with other agents. When an agent needs to communicate with another agent, it performs lookup. If the required agent interface is not found locally, the lookup request is forwarded to the parent agent (which also implements the Service Locator interface), and so on, until the required interface is found somewhere in the hierarchy. After that, the agents interact directly with each other through interfaces, which results in tight coupling among agents. Another issue is that the agents have to implement the Service Locator functionality.

Simple Service Locator mechanism could be used as well. In this case, it should be provided as global system service to all agents.

6 Business Logic

As the dialog control logic glues together the application's presentation and the business logic, it is important to know how the latter is designed, and how it can be accessed. . For example, Domain Model or Transaction Script patterns [Fowler02] could be applied when implementing the business logic. In this section we discuss how it is treated by the patterns for GUI applications.

6.1 Business Logic in MVC

Business logic is handled in MVC by the model component(s). In section 5.1 on the dialog control issues in MVC, we have outlined the problem of what the model actually is. Here, we discuss this issue in details.

In small applications with simple business logic, we may have just one model component. View(s) and controller(s) work with this single model.

In large applications, the underlying business logic is usually rather complex, and involves various business objects and relationships among them. With the Domain Model pattern [Fowler02], a business object is implemented as a separate class or a set of related classes. The following questions arise in this regard:

- What are the MVC model components in applications with complex business logic?
- How do the business objects relate (or could be mapped to) the model components?

A straightforward way of applying MVC is to consider each business object that is displayed to the user as an MVC model and associate it with the corresponding view(s) and controller(s). The main problem with this approach is that we have to design business logic in the MVC way. In particular, we have to put the change notification mechanism (using the Observer pattern) into the business objects, thus mixing the concerns. In some case, business logic may already exist, so it could be highly expensive or even not feasible to add the MVC-specific functionality. We might also have views that display a number of business objects (e.g., trees or lists) implying that these views depend on several models (see also section 5.1). Technically, we must extend the change notification mechanism for such views, so that they are able to find out which of the associated models has fired a change notification event.

In order to avoid having MVC specifics being implemented in the business logic, we can add a layer of indirection. For instance, we can have a separate model component for each business object. In this case, the model holds the data to be displayed, while the business object implements the corresponding business logic. The model forwards user input to the business object. It also performs data type transformations between the presentation and business logic and implements change notification mechanism. Thus, we decouple the business logic from the presentation and (part of) dialog control issues.

Business logic is often hidden behind Façade objects, where each Façade is responsible for one or several (usually related) use cases. In such a case, we can consider Façades as MVC models. In particular, they have to implement the change notification functionality. As Façades are more coarse-grained that particular business objects, each one will have more

views (and controllers) associated with it. So, we might need to introduce event types and let the views subscribe only for events they are interested in. Otherwise, a Façade would notify all its associated views upon each single state change.

6.2 Business Logic in HMVC

HMVC does not assume business logic be implemented in the model components (in contrast to pure MVC). In general, the developers are free in how they implement the models. For instance, the model within an MVC triad can keep the data to be displayed and retrieve data from an application server or a database. Hence, the models decouple the presentation from the business logic (acting as Façades). As an important consequence, the developers are not forced to incorporate HMVC specifics into the pure business logic, which could be implemented the way the developers like.

6.3 Business Logic in PAC

In PAC, the application's business logic is implemented in the abstraction component of the top-level agent, which provides an interface to retrieve and change business data. Hence, we can see it as a Façade to the business logic. On the other hand, bottom-level agents usually represent 'self-contained semantic concepts' [POSA I], that is, business objects from the business logic. The state of the business object is then kept in the agent's abstraction component and is a duplication of the business object's state from the business logic implemented in the top-level agent. Therefore, the abstraction component of the bottom-level agent just holds the data, but does not provide any business operations on the corresponding business object. It must therefore be kept synchronized with the abstraction of top-level agent (see also section 5.3 on the dialog control problem in PAC).

The design of the business logic itself is left to the developers.

7 Summary

The complexity of GUI development is sometimes underestimated in software community, as compared to the server-side programming. In this paper, we have tried to specify common problems when developing GUIs, and how these problems are, or could be, addressed by existing patterns for GUI development. We have also tried to identify the problems when applying these patterns for large and complex GUI applications and to illustrate all this with various examples. A detailed discussion of really complex (and therefore interesting) examples deserves, however, a particular paper.

We do not want to give here any conclusions regarding each particular pattern we have discussed or compare the patterns with each other, but rather end this paper with some general observations.

First, separation of concerns is the main principle underlying each pattern. Second, each pattern works rather well in simple cases. On the other hand, pattern descriptions give only basic considerations, without going much into details. The more complex the GUI application is, the more questions will arise, which are not answered by the patterns, and the more design alternatives the developers will have. Identifying visual component classes in the container-level MVC approach and designing complex presentation parts in PAC are examples of the problems the developers have to solve themselves. Another example is designing the dialog control to provide interaction among different application parts.

For us, the main result of our work on applying and analyzing patterns for GUI development is that there is always place for creative work – to go beyond the pattern descriptions and investigate new design alternatives. And this is in patterns' nature – they do not (and probably must not) provide ready solutions, but rather give an advice on what to start with.

8 References

[Arch92] A Metamodel for the Runtime Architecture of an Interactive System. The UIMS Developers Workshop, SIGCHI Bulletin 24 (1), 1992.

[Burbeck92] S. Burbeck. Application Programming in Smalltalk-80: How to use Model-View-Controller (MVC), 1992.

[CKP00] J. Cai, R. Kapila, G. Pal. HMVC: The Layered Pattern for Developing Strong Client Tiers. www.javaworld.com, 2000.

[Coutaz87] J. Coutaz. PAC: An Object Oriented Model for Implementing User Interfaces. SIGCHI Bulletin, 19 (2), pp. 37-41, 1987.

[Evans03] E. Evans. Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison Wesley, 2003.

[Fowler99] M. Fowler et al. Refactoring: Improving the Design of Existing Code. Addison Wesley, 1999.

[Fowler02] M. Fowler et al. Patterns of Enterprise Application Architecture. Addison Wesley, 2002.

[GoF] E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns. Elements of Reusable Object Oriented Software. Addison Wesley, 1994.

[HO07] M. Haft, B. Olleck. Komponentenbasierte Client-Architektur. Informatik Spektrum, 30 (3), 2007 (In German).

[Holub99] A. Holub. Building User Interfaces for Object Oriented Systems. www.javaworld.com, 1999.

[KP88] G. Krasner, S. Pope. A Description of the Model-View-Controller User Interface Paradigm in the Smalltalt-80 System. Journal of Object Oriented Programming, 1 (3), pp. 26-49, 1988.

[Marinilli06] M. Marinilli. Professional Java User Interfaces. John Wiley & Sons, 2006.

[POSA I] F. Buschmann et al. Pattern-Oriented Software Architecture. A System of Patterns. John Wiley and Sons, 1996.

[Wellhausen2005] T. Wellhausen. Ein Client-Framework für Swing. JavaSPEKTRUM, 2005 (In German).

Patterns for Licensing Web Services*

G.R. Gangadharan, University of Trento, Trento, Italy
Michael Weiss, Carleton University, Ottawa, Canada
Vincenzo D'Andrea, University of Trento, Trento, Italy

Abstract

The trend towards providing software as a service has required us to rethink the way software is licensed. There are different types of proprietary and open source licenses for software. However, the nature of web services differs significantly from traditional software and software components, which prevents the direct adoption of their respective licenses. As web services can be accessed and consumed in a variety of ways, there is a also spectrum of licenses for web services. We have mined existing licenses for web services for commonalities, and present the different licensing options in the form of patterns.

1 Introduction

Increasingly, software is provided as a service. Specific examples include the Google web services, the Amazon cloud computing service, or the StrikeIron data services. Though web services are software, they differ from traditional software in many ways:

- Services are executed in a hosted infrastructure. Consumers do not install applications, and a new class of issues is created by the accessing the service over a network.
- Services are designed to facilitate reuse. Services abstract from language and platform-specific aspects of the underlying software (loose coupling).
- Services encourage composition. Service composition allows consumers to build coarse grained services by combining finer grained services to any level of hierarchy.
- Services are data-driven applications ("data is the next Intel Inside" [1]). We need to distinguish between the use of a service as software, and the use of the data it provides.

These differences prevent the direct adoption of licenses for traditional software and software components [2]. By a license, we refer the terms and conditions that accompany a piece of software or a service. The party defining those terms is known as the licensor. Licensing principles reflect the overall business value of software to its producers and consumers. Licensing is often also used to provide protection to software producers for their intellectual property rights, and thereby becomes a source of revenue and a tool for business strategy. Licenses for web services reflect the differences between traditional software and web services: they govern

the execution, reuse and composition of the services. Although of great practical relevance, licensing of the data provided by services is out of scope for this paper.

In this paper, we classify web service licenses as proprietary or open [3]. Proprietary software licenses allow the execution of the software (including components) in the licensee's computing environment. Open source licenses allow consumers to view, modify, and share the source code and redistribute the software either for commercial and/or non-commercial purposes. However, for web services, we also need to consider their execution/usage. Execution of a web service refers to access/use (invocation) of the web service by another service.

A web service has an interface part, which defines the externally visible functionality (and typically some non-functional properties), and an implementation part, which realizes the interface. The opaque nature of services often hides the details of operations from service consumers. A consumer can be restricted from either seeing anything beyond the interface, or understanding how a service is composed from other services. Which parts of a service are made accessible to the consumer in addition to the (always accessible) interface, and which additional rights the consumer is given, is determined by the terms of the license.

Below, we describe patterns for licensing proprietary services and open service. The relationship between these patterns is shown in the pattern roadmap in Figure 1. The diagram uses circles to represent the common context shared by a group of patterns, and rounded rectangles to represent patterns. This gives us a way to refer to a group of patterns which solve related problems, for example, all the patterns relating to limiting execution. The audience for these patterns includes managers of companies who offer software as a service, and developers who need to understand the business impact of those licenses. Our focus is, therefore, on the strategic aspects of licensing web services, not on the underlying technology.

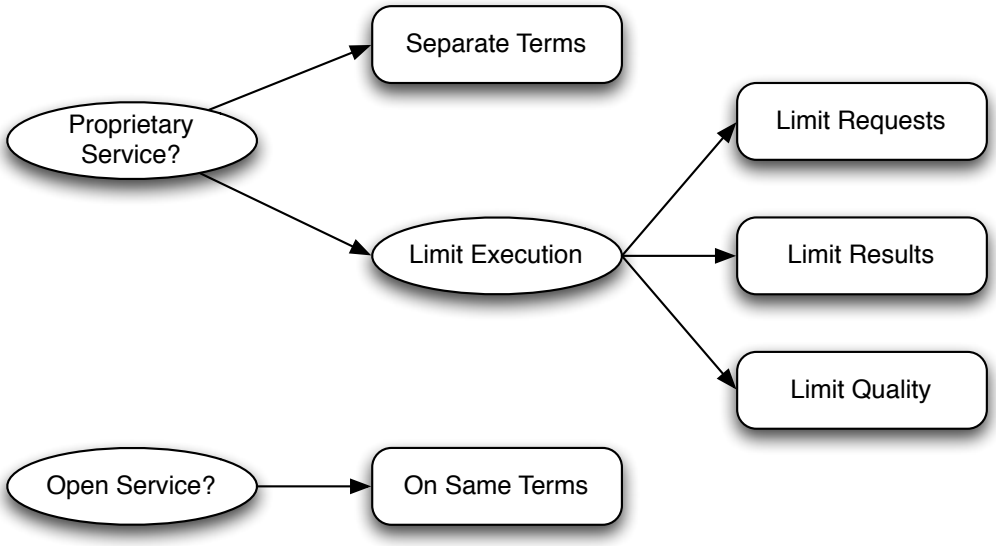


Figure 1: Roadmap for web service licensing patterns

2 Separate Terms

2.1 Example

Banca Indica offers a Daily Exchange Rate web service through which consumers can calculate foreign exchange rates based on the daily rates offered by Banca Indica. It also keeps a historical data record of exchange rates based on Daily Exchange Rate, which can be accessed via another web service, Historical Exchange Rate. Banca Indica offers Daily Exchange Rate as a free service and the Historical Exchange Rate service at 1 euro per use.

2.2 Context

A service provider that offers web services.

2.3 Problem

How can a service provider use the terms and conditions of a service license to attract consumers of its free service to subscribe to its premium service?

2.4 Forces

Free services provide an incentive to users to upgrade to paid-for services. They allow consumers to evaluate the service, before they make a decision to purchase the service. However, the free services should not be sufficient for power users, who would be willing to pay for a more advanced service. It must be beneficial for those users to upgrade. Providing free services is also generally not free for the service provider itself (development costs, hosting fees, sublicenses). Hence, the cost of providing the free service must be carefully balanced against any potential revenue from premium services, and non-monetary benefits (reputation).

2.5 Solution

License free and premium services under separate terms and conditions. For example, free web services can be licensed under one set of terms that specify which services (or service operations of a single service) are offered for free, and under what restrictions they are provided (see also Limit Execution). Paid-for services can be offered under terms that define their usage terms (which are more attractive to power users) and pricing terms.

2.6 Example Resolved

Banca Indica offers the Daily Exchange Rate service under the following terms and conditions:

1. The service cannot be composed with other services.[1]
2. The service can be used without a fee.

Banca Indica also provides an Historical Exchange Rate service under a different set of terms and conditions, which are more attractive to power users:

[1] This example should not suggest that, in general, free services cannot be integrated into other applications. The payment and composition terms of a license are orthogonal to one another.

1. The service can be composed.
2. The service requires a payment of 1 euro per use.

2.7 Consequences

Providing different versions of a service under separate terms gives the service provider control over how the service is used. The restrictions of the free service will not satisfy more demanding consumers, who will want to upgrade to the premium version. Yet, while they are evaluating the service, these same restrictions (eg non-commercial use) may be acceptable. However, when providing a version of the service for free, there is a risk that the service provider overestimates the potential demand, and its offering is satisfactory to all consumers.

2.8 Known Uses

The general Amazon Web Services (AWS) Licensing Agreement[2] states:

> The services covered by this Agreement include both free services that AWS and its affiliates (referred to together herein as "we" or "us") make available for no fee, for the purpose of promoting sales on the Amazon.com website and related websites and for other purposes (the "Free Services"), and services that we make available for a fee (the "Paid Services").

The Amazon Simple Queue Service (SQS) is a premium service provided by Amazon that offers a reliable, scalable and hosted queue for storing messages as they travel between computers. Below, we quote from the pricing options for the Amazon SQS service:

> Pay only for what you use. There is no minimum fee.
>
> **Requests**
>
> USD 0.01 per 10,000 Amazon SQS Requests (USD 0.000001 per Request)
>
> Amazon SQS requests are CreateQueue, ListQueues, DeleteQueue, SendMessage, ReceiveMessage, DeleteMessage, SetQueueAttributes and GetQueueAttributes.
>
> **Data Transfer**
> USD 0.10 per GB - all data transfer in
> USD 0.18 per GB - first 10 TB / month data transfer out
> USD 0.16 per GB - next 40 TB / month data transfer out
> USD 0.13 per GB - data transfer out / month over 50 TB
>
> Data transfer "in" and "out" refers to transfer into and out of Amazon SQS. Data transferred between Amazon SQS and Amazon EC2 is free of charge (i.e., USD 0.00 per GB).

Hence, the general AWS Licensing Agreement lists the terms and conditions common to all free and paid services. Pricing terms for premium services are listed separately.

[2] http://www.amazon.com/AWS-License-home-page-Money/b?ie=UTF8&node=3440661

2.9 See Also

Separate Terms is not specific on how the service provider should select the terms for the different versions of the service. The Limit Requests, Limit Responses, and Limit Quality patterns discuss ways how the service provider can limit service execution.

License patterns for proprietary software have been described by Kaminski and Perry [5]. They can be used by developers to select an appropriate license type for their software.

Segmenting customers into free and premium is an application of Segmented Customer [4].

3 Limit Execution

Rather than a pattern, this section describes a common context for the following three patterns. The common starting point for these patterns is a service provider that uses Separate Terms to attract users to its service by offering different licenses for high-end and low-end versions of the service. Now, the provider needs to devise license restrictions that create an incentive for users to upgrade the service, once they have had a chance to evaluate the service. The service may also not be ready for real deployment, but the provider wants to create attention around the service without creating a wrong impression of the potential capabilities of the service.

The solution involves imposing constraints on the execution of the web service. The patterns in the next three sections suggest different ways how this can be done: Limit Requests, Limit Results, and Limit Quality. These strategies could also be applied in combination.

4 Limit Requests

4.1 Example

A foreign currency exchange service Xenon provides buy and sell rates of large-value transactions in global currency markets. Any registered service consumer can make requests (invocation to the Xenon service) up to 10 times per day.

4.2 Context

A service provider that uses Separate Terms for free and premium versions of its service.

4.3 Problem

How can a service provider use license terms to control the execution a web service?

4.4 Forces

The number of times a service can be "freely"[3] executed should be more than sufficient for light use or use during development, but it should not allow heavy use of the service.

4.5 Solution

Impose constraints on the number of invocations of the web service. These restrictions may include the number of times a web service can be executed, predefined purposes for which it can be used, the type of user (e.g. free to academic institutions), or the level of payment.

4.6 Example Resolved

The Xenon service includes the following terms to restrict the invocation to no more than 10 times per day:

> You should not use more than 10 invocations per day to the Xenon service.

4.7 Consequences

A user who wishes to execute the service under regular load conditions will need to subscribe to a premium version of the service. During development the limited version is sufficient. A possible drawback of the pattern is that, because it is "free", the limited version may attract so much attention that the provider does not have enough capacity to handle the requests.

4.8 Known Uses

StrikeIron offers different payment schedules for its web services: monthly subscription, annual subscription, or one-time purchase. Its service agreement[4] states:

[3] Instead of considering free vs. paid-for version of service, we can apply the same reasoning to a tiered pricing scheme.

[4] http://www.strikeiron.com/info/faqs.aspx

> A paid subscription is an agreement between consumers and StrikeIron to pay a specified amount of money over a specified amount of time, in exchange for a specified amount of accesses (or hits) to the web service. [...] A 'hit' is the term used in the StrikeIron Marketplace to refer to a counter that is decremented every time a subscriber invokes an operation by accessing and activating the web service.

4.9 See Also

This pattern can be used in combination with other patterns that Limit Execution.

5 Limit Results

5.1 Example

GelPub provides access to a set of online articles in the field of Computer Science. Any users can access the service, but for a given query, the number of results displayed are limited to 10 articles. The remaining search results will be available only to paid subscriptions.

5.2 Context

A service provider that uses Separate Terms for free and premium versions of its service.

5.3 Problem

How can a service provider use license terms to control the execution a web service?

5.4 Forces

The amount of data returned by a service for "free" should be sufficient to evaluate the service. However, as the value of the service lies in its data, it should be difficult to replicate the data that the service has. While the dynamic nature of the data could ensure that obtaining the full data is of limited value, even under these circumstances we would like to be able to charge a premium for more complete results than are available for free, or at a low charge.

5.5 Solution

Specify constraints that restrict the amount of data returned by the web service.

5.6 Example Resolved

The free version of the GelPub service restricts the number of results per query:

> You cannot access information beyond the 10th result for any given query.

5.7 Consequences

The number of results returned by the service are sufficient for light use. However, it is not possible for the user to obtain the full data. Hence, the value of the data is maintained. One possible drawback is that limited results may not give potential consumers the impression that the service is also of limited use. These user will not return, or upgrade to the full version.

5.8 Known Uses

The Google Web Services Licensing Agreement[5] is specified as follows:

> You can retrieve a maximum of 10 results per query, and you cannot access information beyond the 1000th result for any given query.

[5] http://code.google.com/apis/soapsearch/

5.9 See Also

This pattern can be used in combination with other patterns that Limit Execution.

6 Limit Quality

6.1 Example

MicroSync offers a on-demand financial web services. It delivers real-time stock quotes to paid subscriptions. MicroSync also provides stock quotes with a 20 min delay to anyone for free.

6.2 Context

A service provider that uses Separate Terms for free and premium versions of its service.

6.3 Problem

How can a service provider use license terms to control the execution a web service?

6.4 Forces

Some consumers may require a service to deliver high quality data. Others may prefer to trade lower quality data for a lower cost of the service. A provider that offers the same high quality service to all its consumers at one price, foregoes the opportunity to charge demanding consumers a premium, and will be perceived as too expensive by less demanding users.

6.5 Solution

Specify constraints that impose different levels of service quality. You can also impose restrictions that affect the quality of the resulting composed service. For example, you may want to restrict the right to publish the results of the service, as a means of protecting your data.

6.6 Example Resolved

MicroSync delivers real-time (high quality) stock quotes to consumers for a higher fee. The license clauses of these service may differ from the license clauses for a MicroSync service that provides delayed (low quality) data. For example, a MicroSync web service delivering real-time stock quotes may deny composition of this web service with other services.

6.7 Consequences

To be able to provide high quality data (for example, on-time delivery of critical data that changes continuously), a provider may have to make considerable investments. Naturally, this creates an incentive to offer those services at a higher price. These services may attract and retain consumers for whom quality is a top priority. However, this also creates a risk that the demand for the service may not be high enough to justify the investments.

6.8 Known Uses

Xignite offers two versions of its stock quote service: XigniteRealTime which provides real-time stock quotes for U.S. equities, and XigniteQuotes which delivers delayed quotes. The

XigniteRealTime service is offered at a higher price (both require subscription). However, XigniteRealTime also imposes a restriction on how the stock quotes can be used:[6]

> You cannot display real-time information on public web sites.

On the other hand, XigniteQuotes can be displayed on a public web site.[7]

6.9 See Also

This pattern can be used in conjunction with other patterns that Limit Execution.

[6] http://preview.xignite.com/xRealTime.asmx
[7] http://preview.xignite.com/xQuotes.asmx

7 On the Same Terms

7.1 Example

Spells is an open web service providing a spell checking operation for words. A new independently executable web service Spells Mirror is created by modifying the interface and implementation of Spells. Spells Mirror provides a functionality to split a given sentence and another functionality to spellcheck a word by reusing the operation of Spells. Spells Mirror is derived from Spells and is value-added as it provides its own additional functionality. Since Spells is an open web service, its license clauses determine the amount of control its creators can exercise over value-added services that are derived or modified from Spells.

7.2 Context

A service provider that offers web services.

7.3 Problem

How do we prevent proprietary lockup of open web services when they are composed?

7.4 Forces

Users should be able to modify a service, or derive new services from the service. However, in order to avoid license forking, we would like to prevent that the new service is licensed differently from the parent service. In this way, the value-added by the changes can benefit the whole community created around the web service. This benefit needs to be balanced against the need of service providers to generate profit from their service offerings.

7.5 Solution

Include a condition in your license that derivations of the service or modifications must be licensed under the same terms (a sharealike clause). A web service license with a clause similar to the "ShareAlike" clause of the Creative Commons license requires value-added web services to be licensed under the same terms and conditions. These clauses prevent others from turning value-added web services into closed services, if the parent web service is open.

Opening a web service means making the source code of the service implementation available in addition to the source of the service interface.[8] Inspired by how open source software is licensed, an open web service allows access to the source code of its interface as well as its implementation, allowing freely distributable composite and derivative services.

An open web service can expect another web service that uses ths service to reflect the same terms and conditions. Though open source software licenses do not discriminate among the uses of a software, the dynamic binding and execution of web services can enforce certain restrictions on the execution/usage of open services similar to Limited Execution.

7.6 Example Resolved

If Spells is released with a license clause that includes a sharealike clause, then

[8] A web service interface is, in a trivial sense, always available.

- Spells Mirror should be an open web service.
- Spells Mirror should be licensed under the same license as the one that Spells has.

Under this scenario, any value additions to the open web service Spells remains open, thus benefitting the community and avoiding forking of the license.

7.7 Consequences

Sharealiking (ie specifying a sharealike clause in the service license) prevents license forking, and benefits the community by returning user contributions to the community. However, another service provider could build a value-added web service based on a different open web service with more or less similar functionality. In this case, the parent web service may fail to retain users. Hence, using a sharealike clause carries the risk that it is perceived as too strong, and it may motivate others to derive from services that are less restrictive.

7.8 Known Uses

The GNU Affero General Public License[9] (AGPLv3) is a free, copyleft license for software and other kinds of works, specifically designed to ensure modifications or derivations of software are returned to the community in the case of network server software (ie a service).

> You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:
>
> [· · ·]
>
> d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.

WikiDot[10], a farm of Wiki sites, uses the GNU AGPLv3. It requires releasing any changes to the service implementation. Funambol[11], a leading provider of mobile 2.0 messaging software built on open source stacks, offers its services under a GNU AGPLv3 license.

7.9 See Also

Open source license patterns for traditional software have been described by Kaminski and Perry [6]. The rights granted in an open source software license range from basic access to the source code of the software to the rights to make copies and distribution of the software.

[9] http://www.fsf.org/licensing/licenses/agpl-3.0.html
[10] http://www.wikidot.com
[11] http://www.funambol.com

8 Acknowledgements

Our sincere thanks go to our shepherd, Tim Wellhausen, for his probing comments.

References

[1] Musser, J., O'Reilly, T.: Web 2.0 Principles and Practices. O'Reilly Radar. O'Reilly (2007)

[2] D'Andrea, V., Gangadharan, G.R.: Licensing Services: The Rising. In: Proceedings of the IEEE International Conference on Internet and Web Applications and Services (ICIW'06), Guadeloupe, French Caribbean. (2006) 142–147

[3] Merges, R., Menell, P., Lemley, M.: Intellectual Property in The Technological Age. Aspen Publishers, New York (2003)

[4] Kelly, A.: More Patterns for Technology Companies Product Development. In: Proceedings of the European Conference on Pattern Languages of Programs (EuroPLOP). (2007)

[5] Kaminski, H., Perry, M.: Pattern Language for Software Licensing. In: Proceedings of the Tenth European Conference on Pattern Languages of Programming (EuroPLoP). (2005)

[6] Kaminski, H., Perry, M.: Open Source Software Licensing Patterns. In: Proceedings of the Sixth Latin American Conference on Pattern Languages of Programming (SugarLoafPLoP). (2007)

Using a Profiler Efficiently

Strategies that Help you to Find Performance Problems and Memory Leaks

Tim Wellhausen

kontakt@tim-wellhausen.de
http://www.tim-wellhausen.de

May 24, 2009

Proceedings of the 13th European Conference on Pattern Languages of Programs (EuroPLoP 2008), edited by Till Schümmer and Allan Kelly, ISSN 1613-0073.

Abstract: Sooner than later most software development projects suffer from severe runtime problems. When features are given top priority, caring for non-functional requirements such as performance or stability is most often postponed during the initial development phase. Once a system is in production, however, performance problems and memory leaks quickly catch more attention. A Profiler is a very useful development tool to find the causes of these problems. Using a Profiler is not that easy; you need good strategies to detect the actual causes. This paper gives you advice how to use a Profiler efficiently.

Introduction

Performance problems and memory leaks are encountered in many software development projects. Unfortunately, they often have subtle causes that are not apparent by introspecting the code. In particular, complex, multi-layered software systems are hard to debug to find these causes.

Just as a Debugger is the tool at hand to track down problems that affect the correctness of a software system, a Profiler tool can be very useful to trace performance problems and memory leaks. Only a Profiler gives you an accurate view on what's happening inside the system either over a period of time or at a specific point of time.

A Profiler supports the analysis of a software system at runtime: (1) by finding the causes of real or perceived slowness of a system, i.e. those parts of the system that consume more time to fulfill a functionality than they should take or (2) by finding the causes of memory leaks that exhaust the available memory until the application runs very slow or stops running at all.

This paper presents usage patterns that cover both aspects of using a Profiler. The patterns are independent of a specific Profiler product. However, there are some products that support all usage patterns, whereas other products only support some of them. This paper neither gives you an overview of the available products nor explains their completeness regarding the patterns.

Note that this paper assumes that you are already familiar with the Profiler tool of your choice, i.e. that you know how to start a Profiler session and how to take a memory snapshot, for example. Also note that this paper only addresses Profiler products with a rich graphical user interface; Profilers that only record data in text form are out of scope.

Although the patterns are not dependent on specific technologies, they are based on experiences in profiling object-oriented, single- or multi-layered software systems that are developed on a technological platform that involves garbage collection at runtime (for example Java and .NET). It has not been deeply analyzed yet how valid the patterns are if applied to other programming languages and platforms, in particular to the area of embedded software.

The patterns are presented one by one. Each patterns has a short problem and a short solution statement, written in bold font. To get an overview of the pattern, you may first just read these statements for each pattern. Then, you may read the patterns one after another or you may start by reading the first pattern, THINK ABOUT IT FIRST, and then follow the recommendations as given in the patterns' descriptions.

At the end of the paper, you can find an Examples section that shows how the patterns can be applied in sequence, illustrated by real world applications. After that, you can find references to other resources about profiling.

Think About it First

You are responsible to analyze a software system to detect the cause of its severe performance problems or memory leaks. Maybe you always have a good gut feeling of possible causes for such problems; but so far, you don't know the causes yet.

How do you start solving the performance or memory problems of a software system?

As a software developer you are accustomed to find solutions for given problems. If you don't know the exact reason for a problem, you might, for example, be tempted to start developing a solution that incorporates well-known design patterns to improve the performance of your software system in general, such as an object cache or a resource pool.

But without knowing the actual causes, you cannot be sure that any changes you perform on the software system actually improve the performance or remove the memory leaks. Whatever you are doing might simply add complexity to your software system but may not improve it. Or even worse: may introduce new bugs to previously running code.

Therefore:

Don't guess what the causes of the problems might be because quite often you're going to be wrong. Instead, use a Profiler to analyze your software system!

What sounds like a mundane advice already is the single most important advice this paper has to offer. If you don't know the actual reason for a performance problem, don't be induced to prematurely start coding a solution, even if you believe that this solution might solve the problem. More often than not, the real troublemaker is more subtle than you might think on first sight.

By using a Profiler you can double-check whether your assumptions are right. If are are right, go on and develop the solution you had in mind. If you are wrong, however, be relieved that you have spared yourself from unnecessary work and that you have saved the system from unnecessarily adding complexity.

If you are not the developer who has originally written the software system, chances are that you do not know the complete functionality of the system. In this case, there are hopefully test documents that contain step-by-step instructions for each essential use case or process of the system. Having such instructions at hand, you can more easily profile the system to understand both its functionality and runtime behavior.

Naturally, there are cases when your gut feeling is right. If you never trust your guesses but always check first, you might lose time fixing your software system. So there is a trade-off to make when time is the most limited resource at hand. In this case, it may help to time box any efforts following your gut feeling. Additionally, learning and applying tools is much effort that you need to afford in your project.

If the system runs slow, you should begin the profiling session by Finding Performance Anomalies. If you guess that the system suffers from memory leaks, Check for Memory that the software system consumes at runtime.

Find Performance Anomalies

Your software system runs slower than you think it should. The normal workflow seems to be okay but some deviations happen. You don't know yet why.

How do you start tracking down performance problems?

Some performance problems may materialize at places and at times that are not obviously connected to their real causes. If you delve down into details at once, you may miss the actual causes and get lost in too much information that does not lead you anywhere. Getting lost in such a way often is frustrating and may make you stop profiling and start guessing what the reasons might be.

If you don't know the software system very well, you might not easily distinguish between acceptable and unacceptable performance. Some parts of the system behave better than others. How do you know which behavior is good enough and which is not acceptable any more?

Randomly trying to perform some actions to get an impression of the system performance is like looking for a needle in a haystack. You can easily spend a lot of time without getting any hints where to look closer.

Therefore:

First get a general impression of how fast typical operations execute. Then try to find anomalies by comparing more specific actions with these numbers.

Only seldom do all parts of a software system suffer equally from performance. Skimming over many parts of a system should give you a good impression of the overall characteristics you might expect. You could, for example, set the fastest non-trivial operation as a benchmark for all other operations. Those operations that differ considerably from this benchmark are the best candidates for closer inspection.

The more often you try to find performance anomalies, the better you get to know the performance characteristics of your system. By doing this on a regular basis, chances are better to more quickly spot and improve performance problems.

Always observe the size of the data set that your system operates on. For a realistic comparison, the data set size of several operations should be roughly equal.

Getting a good general impression may be difficult if the software system behaves inconsistently, i.e. differently at different times, in particular if there are active background threads running. Also, if the system contains several performance problems at the same time, it may be very complicated to judge how fast typical operations should execute.

By comparing performance characteristics you should be able to identify at least some parts of the software system that you need to inspect more closely. As next step, you should Isolate Actions.

In case the system does not reveal performance problems under normal load: Stress it. Because using a Profiler slows down the machine on which the Profiler and maybe also the system itself are running, you should try to Minimize the Profiler's Overhead of using the Profiler.

If it is difficult to reproduce performance problems on a local workstation, Profile the Real Thing; if you are not allowed to profile in a production environment, Clone Production. In both cases, try again to Find Performance Anomalies.

Check for Memory

Your software system behaves unreliably. You don't know yet why.

How do you find out whether the software system suffers from memory leaks?

There might be many reasons why a software system does not behave as it should. Some reasons are internal to the system (e.g. a memory leak), some are external (e.g. hardware failure). Before you make any changes to your system you need to be sure that the problem is caused by an internal error.

A memory leak may manifest itself by an obvious system event like an `OutOfMemoryException` in case of Java. But in particular if the system has a lot of memory available, it may take a while until the memory has run up and such obvious messages are shown.

If your system is interactive, use the software application as a user would do. If the system executes batch jobs, manually trigger jobs as they would normally run. In both cases, closely watch the system's memory consumption.

If the system actually has a memory leak, you should notice that the memory consumption of the system increases over time. Be aware that a Profiler typically shows the memory consumption as the size of all objects that are currently alive. Some of these objects may not be in use any more. Therefore, you should watch the activity of the garbage collector and, if necessary, regularly trigger a garbage collector run to watch the size of objects in use over time.

Even if the memory consumption increases over time, this must not necessarily indicate a memory leak. Other causes might be data caches that fill up over time, additional code that is loaded at runtime, data that is stored in a session as long as a user is logged in, or data sets whose size increase.

You therefore need to have a good general knowledge about the software system to be able to distinguish between acceptable and unacceptable memory increases.

Once you know that the system has a memory leak, you need to Isolate Actions that are responsible for the memory leak. If the memory leak causes a proliferation of objects, chances are that the Profiler puts your local system under heavy load while keeping track of what's happening inside your system. In that case, Minimize the Profiler's Overhead.

Minimize the Profiler's Overhead

You intensively use a Profiler on a local workstation to track down performance or memory problems.

How can you avoid that the Profiler itself negatively affects the application?

In a real-world application, many operations are executed in a short period of time. Within a couple of seconds, millions of objects may be created and disposed, and thousands of operations may be called. Because the Profiler cannot guess the reasons of the problems at hand, it has to collect all available information to present to you the most accurate view of the internals of the software system.

To monitor a software system in every detail, the Profiler itself consumes many resources, i.e. it needs a lot of memory and takes a considerable amount of time. Using a Profiler may therefore consume so many system resources that the software system to analyze is negatively affected. Connections may time out, network packets may be dropped, or the graphical user interface may become too slow to use.

Therefore:

Before you start a profiling session, reduce the amount of information gathered by the Profiler to the absolute minimum.

Most Profiler products provide a multitude of options to control the behavior of the Profiler itself. As most Profiler products support both the analysis of memory consumption and the analysis of runtime performance, they also support selectively turning these features on and off.

Some more sophisticated Profilers give you detailed control of the granularity with which the Profiler acts. This means, for example, if you need only a general impression of the memory usage, it may not be necessary for the Profiler to record every object creation but to take snapshots of the memory consumption every now and then.

Quite a few Profiler products provide the possibility to start and stop the collection of data at runtime. This means, you are able to initially turn off most options to let the software system start without interference by the Profiler. As soon as the system is ready for profiling, you can selectively turn on the collection of required information.

The more settings the Profiler product provides the better you may thus minimize the overhead of monitoring the software system. But this also mandates that you are aware of what exactly you need to know about the software system. If you turn off some options to gather information, maybe you miss exactly those information that would help you to understand the cause of the problems. In particular, if changing these settings dynamically is not possible, it may be cumbersome to restart the Profiler to change the settings and try again.

Another way to minimize the impact of the Profiler itself may be to Isolate Actions and to turn on the profiling only for performing the actions that you would like to analyze isolated from all other actions.

Isolate Actions

Your application has performance problems or memory leaks and you've got a good general impression of the runtime behavior of the whole software system.

How can you track down the reasons for the problems at hand after you got first indications of what they are?

Because a Profiler can give you a wealth of information, you may easily get lost. In particular, if you more or less randomly execute some functions of the software system, you will have a hard time to isolate the problems.

Therefore:

Follow a Divide and Conquer strategy, i.e. perform distinct actions, preferably small steps at a time, and check the outcome of the Profiler after each action or step.

You first need to come up with a sequence of actions during which you assume that the problem takes place, i.e. during which the memory consumption increases significantly or the execution time of the operations is far too long. After each step, check the data the Profiler presents to verify that the step performs as it should be.

By pursuing a divide and conquer strategy, you may start with more coarse grained steps until you find a peculiar step. Then split this step into several smaller steps and repeat these steps until you find that step that is responsible for the problem.

Be aware that caches and background threads may alter the results of successively performing the same operations. If that is possible, disable these caches and background operations while trying to isolate the actions that cause the actual problems.

This pattern can only be applied for actions that can easily be executed repeatedly. In particular if some problems only appear during the system's startup or in the last seconds before the system crashes, it is very difficult to isolate them.

You should also always be aware that you may drill down into the software system at the wrong location. If you realize that you analyze the wrong part of the system, track back and start over. If you do this several times in a row, stop the Profiler and think over your assumptions. It may help to either Find Performance Anomalies or Check for Memory again.

If you are tracking down a performance problem, it may help to Repeat Actions to get more significant numbers. If you need to analyze a memory leak, try to Come Full Circle to more accurately compare memory snapshots.

Repeat Actions

The software system to profile has subtle performance problems. You have some candidate actions in mind that probably cause the problem.

How can you clarify the cause of performance problems?

Even if you are able to isolate suspicious actions, those operations that are actually too slow may not be obvious. It could be, for example, that some operations have a big up-front initialization overhead that tamper the results. It could also be that some operation have a higher intrinsic complexity than others, which does not stick out clearly.

Some software systems do not behave the same every time the same operation is executed. Depending on a lot of different factors, a software system may be slowed down temporarily. The reasons may be worker threads in the background such as the garbage collector, event processing, etc.

Therefore:

Execute the candidate actions multiple times in a row to suppress side effects and to magnify the actions' effects on the system's performance.

If you execute the same action several times in a row, side effects such as background work or initialization effort diminish against the actual complexity of an operation. The more often you repeat an action, the less weight statistical mavericks have.

Repeating an action may be achieved by manually starting the same action over again or by letting the action be executed automatically. If a software system needs to evaluate lines of a text file, for example, you could provide a bigger input file. In particular repeating an algorithmic operation often gives you a good view on its intrinsic complexity.

This pattern can easily be applied if the software system has a graphical user interface with which you may start and restart an operation without significantly changing the state of the system. If you need to perform heavy-weight actions to reset the system and execute again the action, this may already tamper the performance evaluation too much to get meaningful results. This pattern can also not be applied successfully if the cause of the performance problem is part of the initialization work or if the problem cannot be reproduced because it relies on side-effects.

If you repeat actions but still cannot clearly identify the reasons for the performance problems of the software system, you could Stress it or try again to Find Performance Anomalies. It may also help to drill down into the actions that you have repeatedly executed and try to Isolate Actions again.

Come Full Circle

The software system to profile has a memory leak and you have been able to isolate the action that causes the leak.

How can you identify the actual objects that cause a memory leak?

A memory leak appears when some memory has been reserved but not set free after its usage. This means, every memory leak is caused by an operation that reserves memory, holds it, and does not set it free when it is no longer needed. You must therefore find this operation.

Every operation that the software system executes may change the memory consumption. By analyzing an arbitrary snap shot of the memory consumption, it is hard to tell which objects are in regular use and which objects should have been given free earlier.

Therefore:

Find a sequence of actions that includes the offending action and that leaves the software system in the same state as before. Then compare the number of living objects of the same classes and identify those that have been increased but should not have.

If the software system has a graphical user interface you may be able to open a dialog, execute an operation and close the dialog again to free all resources needed by the dialog. If the memory leak appears in a part of the software system for which no graphical user interface exists, you may trigger system jobs that execute the actions and leave the system in the same state as before.

In each case, you need to mark the state of memory consumption of the software system as reference before you start to execute the sequence of actions. After the sequence of actions is executed, you may compare the state of the memory consumption to that reference. Many Profiler tools provide a function to set a marker against which the memory consumption is permanently compared.

To be successful, you need a close understanding of the objects involved in the actions because you need to find those objects that do still exist at the end of the sequence but should not exist any more. Depending on the platform, the programming language, and the settings of the Profiler, you may need to explicitly trigger a garbage collector run to remove all unused objects before you can analyze the memory consumption in detail.

Applying this technique is very difficult if the software system changes its internal state and therefore changes its internal memory consumption during the execution of actions that come full circle. You then need to check very closely which objects may still exist and which objects may not. It might also be necessary to stop any work done in parallel on background threads or by asynchronously started jobs.

If you have found the objects that actually cause the memory leak, you may still not know the reason for their existence. Try to Trace the Roots to find the reason why they have not been removed from memory. If you're stuck because your assumptions about the memory leak have mislead you, try again to Check for Memory.

Trace the Roots

You have found a memory leak, i.e. identified objects that still exist in memory when they should not exist any more.

How do you find the reason why offending objects are still in memory?

It may be straightforward to find objects that are still alive but should not. But the pure existence of these objects does not explain why they still exist. There may be many places in the source code where these objects have been created, they may have been passed as parameters to many methods, and they may be referenced from many other objects.

Even if you have identified the objects by coming full circle, maybe many steps were necessary to return the system to the same state as in the beginning. Manually introspecting the source code that has been executed by all of these steps may just not be feasible.

Therefore:

Create a snapshot that includes all living objects and pick a single object that should not exist any more. From this object on follow the incoming object references until you reach the root object that is responsible to hold the whole chain of objects.

To apply this pattern, you need a Profiler tool that has a graphical view on the network of interconnected objects of a memory snapshot. This means, you should be able to choose any living object and expand all of its incoming object references graphically. You need to recursively check the incoming references of all referring objects until you find an object that is valid to exist.

If your technological platform incorporates a garbage collector there are always garbage collector roots, i.e. static objects that may never be removed. All objects that can be reached by object reference chains from these roots are also never removed from memory. You therefore first need to identify the chain of object references from the offending object backwards to a garbage collector root. Then you need to analyze this chain from the garbage collector root on to find the first object reference that should not exist any more. For example, on the object reference chain, there may be a list that should be empty but still contains object references. The actual cause of the memory leak in this case is the code that has not properly emptied or disposed the list.

Manually searching for garbage collector roots may be a very challenging task. Some Profilers provide an option that performs the search for the garbage collector roots automatically. Using such an option, the Profiler tool presents the chains of object references you are looking for. Note that quite often there is not only one such chain but several such chains. You should therefore always first search for a single chain, analyze it, discover the actual bug, fix the bug, and restart both the application and the Profiler to determine whether there were multiple causes that need to be fixed separately.

The fewer connections the software system has at runtime, the easier it is to use this technique. Having a clearly structured system with defined dependencies between internal layers helps a lot to cut down on the number of interconnected objects and therefore significantly reduces the effort to find garbage collector roots. Some software systems, in particular rich or fat client applications typically have a huge number of object references. Manually searching for garbage collector roots is very difficult in these cases.

Stress it

Your software system behaves well when you test it but loses performance under high load.

How do you find performance problems that do not appear when your software system runs in normal operation?

Some performance problems are caused by ill-designed algorithms. These problems can typically be found more or less easily by Repeating Actions. Once you have picked these low-hanging fruits you need to address performance problems that do not always manifest because they may be the results of many interconnected causes that only appear under high load.

You could try to Profile the Real Thing to find the performance problems in the real production system. But quite often this is not feasible: The system may not be ready for production yet, or it is too critical to risk profiling it in the production environment. But without real users, the system may still behave nicely.

Therefore:

Employ a tool that artificially creates a high load on your software system by simulating multiple simultaneous user actions while you profile the system.

There are many tools available to stress test a software system, may it be a rich-client or a web-client application or a system without a graphical user interface. The common denominator of all of these tools is that they are able to simulate the behavior of typical users and that they provide the option to easily scale the number of simultaneous user requests on your system.

Using such a tool, you must first try to identify the typical behavior of the users of your software system. Analyzing the logs of the system or just asking some users may give you an impression of their typical usage. Then you need to simulate and automate the user actions so that they can be replayed by the tool. By slowly increasing the number of simulated simultaneous users you may find the threshold from which on the system does not behave as desired any more.

This technique may reveal performance problems that only manifest in the production system otherwise. However, it cannot reproduce all problems. Some problems are caused not only by a high system load from many users in parallel but by specific properties of the production environment such as the operation system or the server hardware. In these cases, you should Profile the Real Thing or Clone Production and stress test the software system running in a production environment.

If you simulate too many distinct user actions at once, it may be difficult to track down the causes of the problems under high load. In that case you could Isolate Actions and Stress It again, now executing fewer actions at the same time.

Profile the Real Thing

Your software system suffers from performance problems in the production environment.

How do you find performance problems that you cannot reproduce on a local developer's machine?

Setting up a Profiler on a local developer's machine is easy and straightforward in most cases. Still, most often a local machine is configured differently from a production environment, i.e. on the production environment a different operation system may be installed, operation system settings may differ, or there may be more memory or disk space available.

Furthermore, some problems may be caused by the environment in which the production machine runs, in particular causing slow network connections, latency problems, or blocked reverse DNS lookups. All of these differences may be the reason for performance problems that do not manifest elsewhere.

Therefore:

Take on the effort to install and set up a Profiler tool in the production environment and run your tests there.

There are two options to perform profiling tests on the production environment. The more intrusive option is to locally start a Profiler tool and to let it remotely connect to the productive software system. This allows you to deeply analyze the behavior of the software system while it is running.

The less intrusive option is to install a Profiler tool on the production machine that runs there locally, measures the productive software system, and stores information about the runtime behavior in the local file system, for example in flat files. This means, the profiling information is not evaluated at runtime. Instead, you need to periodically get the profiling information and start the graphical user interface of your Profiler tool on your local machine to analyze the collected information. While this option is easier to sell to operators, it prevents you from selectively performing actions and analyzing the systems' behavior.

This advice probably is the most difficult to follow because in many companies there is a strict separation between developing and operating a software system. You may need to convince managers from other teams to allow you to profile your software system in their production environment and you may need to convince the operators whose support you need to actually run any tests. Both tasks may be impossible to achieve. Still, only in the production environment you may be able to analyze problems that appear exclusively there.

After successfully setting up the Profiler tool in the production environment, you should Find Performance Anomalies to get an impression of the runtime behavior of your software system on the production environment. As alternative to profiling the software system in the production environment, in particular if you are not allowed to install a Profiler tool there, you may try to Clone Production. If the performance problems are caused by high load rather than by specific settings of the production environment, stay with profiling a development system and Stress it.

Clone Production

Your software system suffers from performance problems in the production environment.

How can you profile the software system when you must not install a Profiler tool in the production system?

Some problems only appear in the production environment and cannot be reproduced on a local developer's machine. Profiling the software system locally therefore does not help, profiling in the production environment, on the other hand, is not allowed.

You could try to use profiling techniques that do not depend on a Profiler tool, for example gathering as many information as possible in log files. Although it is generally a good idea to write as many relevant information as possible into log files, you cannot retrieve everything of interest from inside the application itself. Besides, changing existing code that has been tested to gather profiling information may not be a good idea as you may have to test again the complete software system. Also, you may not be able to deploy changed code at will but only according to a release plan.

Therefore:

Set up a dedicated test environment that is a clone of the production environment, i.e. that runs on the same hardware and uses the same operation system settings, and profile the software system there.

To set up a test environment, you probably need management support. Typically, the test environment cannot be set up by the development team and setting up a clone of the production environment is expensive, in particular if exactly the same hardware should be used as in the production environment.

To reduce the costs of cloning the production environment, you could try to scale down the test environment without changing the overall characteristics of the system. You could achieve this, for example, by using fewer processors (but still having more than one) or by reducing the available memory.

A clone of the production environment has many benefits other than easier profiling. Most development projects already have distinct environments for development, test, and production. In that case, you should employ the existing test environment for profiling. You probably need to coordinate any profiling tests efforts with functional test efforts carried out by the test team.

A common problem in cloning the production system is the availability of production data. As this data must often be protected from public access, it might be necessary to create an anonymous clone of production data that obfuscates the original context.

The biggest disadvantage of cloning the production system is the effort of keeping the clone in sync. After the initial effort to set up the cloned environment, you need to reflect all changes applied to the production environment. If not, you may not be able rely on the results from testing on the cloned system any more.

After successfully setting up the Profiler tool in the cloned environment, you should Find Performance Anomalies. Cloning the production system alone may not reveal the problems of the software system if these problems do only appear under high load. In that case, Stress it.

Unfinished Patterns

There are more patterns on how to efficiently profile a software system than have been presented so far in this paper. This section gives an overview of patterns that have not yet been elaborated in detail.

Reduce Moving Parts

How do you reliable measure the current performance of your software system?	Stop all background threads and schedulers that might start new threads.

Design for Profiling

How do you facilitate profiling your software system?	Design the system in such a way that profiling parts of it independently becomes possible, for example by employing a layered and component-based architecture and by making it possible to stop background work.

Have a Mental Map

How do you improve your ability to quickly find performance problems and memory leaks?	Try to always have a mental map of the complete system and compare the results of all profiling operations with your expected outcome.

Let the System Warm Up

How do you avoid side-effects when measuring the system performance?	Let the system warm up before you start any measurements so that, for example, all caches are filled with data.

Act as a User Would Do

How can you increase the chance to detect performance problems and memory?	Operate the system as a real user would do, i.e. execute complete use cases without following shortcuts.

Bring Real Users in

How can you increase the chance to detect performance problems and memory leaks if you don't know what the users are doing?	Bring in real users and let them operate on your test system while you closely watch the system's behavior.

Strangle the System

How can you stress your system if it still works quite well under high load or if you cannot produce a very high load?	Limit the resources that are available to your software system, for example, by removing memory or CPUs.

Examples

To relate the given patterns to the real world, this section gives some examples of how the patterns can be applied. The first example shows how to track down memory leaks, the second how to find performance bottlenecks.

Both examples are based on real, open-source applications. Please note that the bugs that are going to be found in these applications have been artificially introduced for the purpose of this section. They have never existed in the original distributions.

Tracking down a Memory Leak

The first example is based on the application *FreeMind* (freemind.sf.net), which is a free and open-source mind mapping tool. FreeMind is a fat client application, developed in Java.

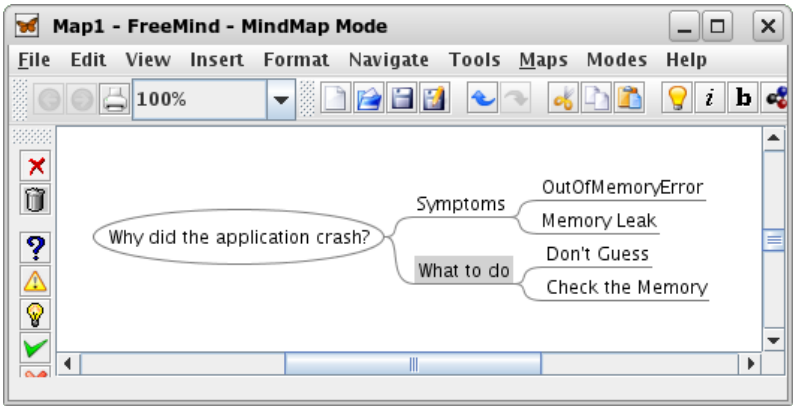
Assume that you are the developer of FreeMind and that you've just got a bug report from a user. The user complains that the application crashes after editing a mind map for some time. This seems to be a serious problem, so you decide to track down its cause.

THINK ABOUT IT FIRST

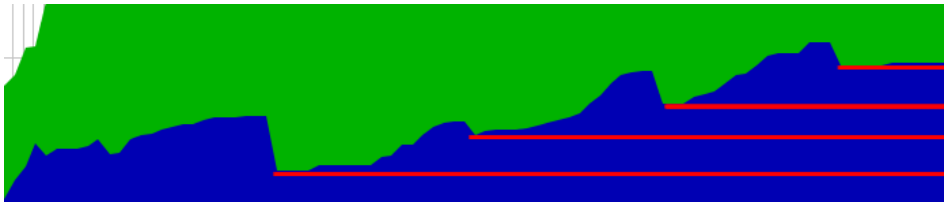
The user also sent you a stack trace of the application that shows an `OutOfMemoryError`, which is an indication that the application probably suffers from a memory leak. You know that there are some cases where you have not properly cleaned up objects before disposing them. But instead of guessing, you decide to start the Profiler to have a closer look to avoid introspecting and maybe changing code that is not responsible for the problem at hand.

CHECK FOR MEMORY

As the user did not report a specific action that he or she performed just before the crash, you decide to first get a general impression of the runtime behavior of the application. Started under the control of the Profiler, the application comes up and you begin to draw a mind map.



You haven't noticed any problems so far, the application runs smoothly; the problem does not seem to appear immediately. However, a look at the memory consumption reveals that something went wrong.



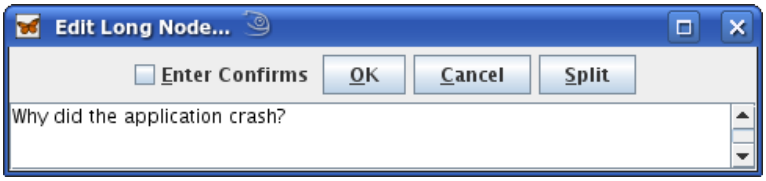
For a reason you don't know yet, the memory consumption grew steadily. The garbage collector ran several times, freeing resources that were not needed any more, but after each run, the used memory increased. To get any further, you need to understand better which action causes the leak.

ISOLATE ACTION

Therefore, you start to selectively perform actions to edit the mind map and closely check the memory consumption after each step. To check the actual memory consumption at a time, you trigger the garbage collector after each step. After a short time you suspect that editing an existing node of the mind map might cause the problem. But you need to verify this assumption.

COME FULL CIRCLE

You assume that the dialog to edit a node might be the cause of the problem. To reliably check the memory consumption of editing a node, you decide to measure the difference before opening and after closing that dialog.



In order to compare the memory consumption, you first trigger another garbage collection run and then mark the current values. After that you open the dialog, edit some text and close it. After triggering another garbage collection run, you have a close look at which objects do now exist that did not exist earlier on.

Among many other objects that don't look suspicious you discover that there are still references to the dialog class that you've just used to edit the node.

sun.awt.X11.XDialogPeer	13	+1 3.744 by...
java.lang.Object	1.353	+1 10.824 b...
javax.swing.JViewport$ViewListener	17	+1 272 bytes
javax.swing.JDialog	13	+1 4.888 by...
sun.awt.X11.PropMwmHints	15	+1 360 bytes
sun.awt.X11.XInputMethod	15	+1 960 bytes
freemind.view.mindmapview.EditNodeDialog$1	13	+1 312 bytes
freemind.view.mindmapview.EditNodeDialog$5	13	+1 416 bytes
sun.awt.X11.XContentWindow	15	+1 1.920 by...
javax.swing.JToggleButton$ToggleButtonModel	21	+1 840 bytes
freemind.view.mindmapview.EditNodeDialog$6	13	+1 208 bytes

In particular, there is now one object more of the class than before. Now you know the reason for the memory leak: a dialog object has not been removed from memory after the dialog window has been closed. Nevertheless, you don't know yet why this happens.

TRACE THE ROOTS

To further understand the problem, you decide to analyze the object graph to find out which objects still hold references to the dialog object. Instead of manually checking all incoming object references to a dialog object, you utilize the Profiler's function to search for garbage collector roots. After a short time, the Profiler shows the first path to such a root.

The graph tells you that the dialog still exists in memory because it is referenced by its native peer object. Because you know the basics about developing a dialog with Java's GUI library Swing, you are sure that somewhere in your code, you forgot to properly dispose the dialog after it is closed.

You switch back to your IDE, open the source code of the dialog class into an editor and find the right spot where you made the dialog invisible instead of properly disposing it.

```
okButton.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    eventSource.setValue(BUTTON_OK);
    dialog.setVisible(false);
  }
});
```

You are relieved to have found the bug so quickly, immediately create a new version of the application and send it to the user that reported the bug.

Discovering Performance Bottlenecks

The second example is based on the application *blojsom* (blojsom.sf.net), which is a free and open-source blog software. Blojsom is a web application, developed in Java.

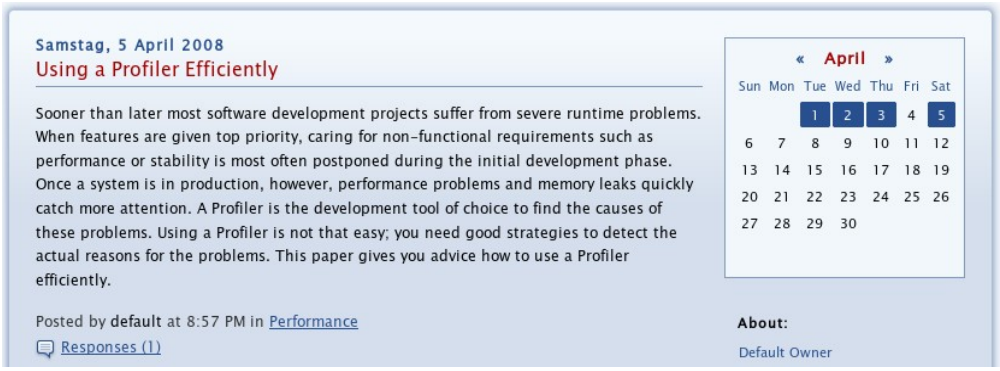
Once again, please assume that you are a developer of blojsom. You have worked hard to finish a new version and are almost ready to publish it. As a last step, you use the software as a normal user would do to find any obvious bugs that have gone unnoticed before.

THINK ABOUT IT FIRST

The application seems to be slower than usual. Maybe some changes you did have deteriorated the performance. Maybe it's just because of the new features that you implemented. One feature in particular could be improved by adding a cache to prevent some unnecessary database calls from happen. But because you want to avoid unnecessary work, you decide to have a closer look before making any changes.

FIND PERFORMANCE ANOMALIES

You set up a new and clean database without any prior blog entries and start up the web server. Still without using the profiler, you create a few categories on the admin pages and then write some blog entries and comments.



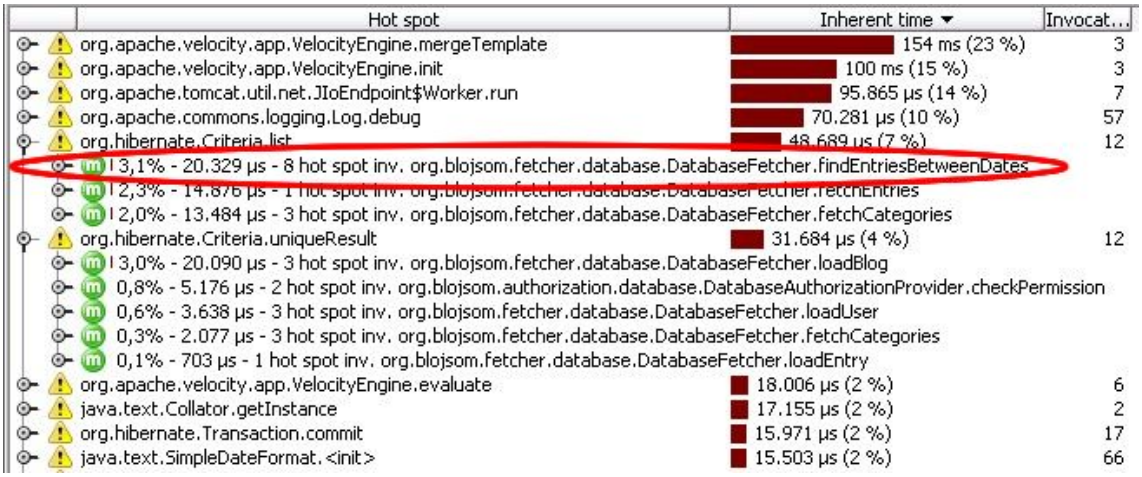
You realize that while the administration pages work as usual, blog pages seem to load slower than before.

Minimize the Profiler's Overhead

Because the Profiler tool may affect the runtime performance of the application, you want to minimize the side effects of using the Profiler tool. You start the tool and change settings so that the Profiler only records performance data and nothing else. You are quite sure that the problems are not caused by the initialization of the application; therefore you start the server with no profiling at all and wait until the application is properly initialized.

Isolate Action

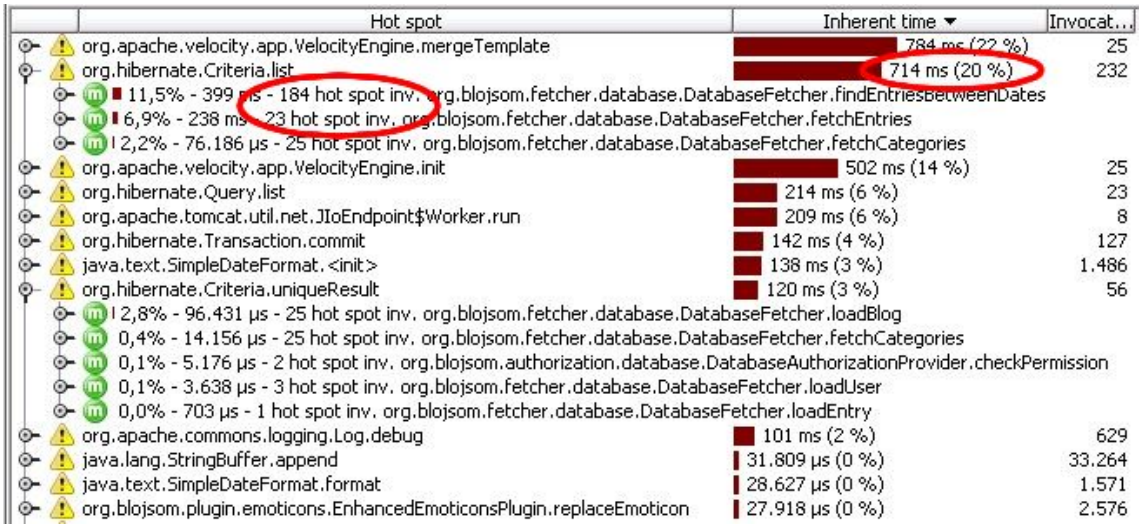
You don't want to get lost in too many details too soon and therefore decide to perform several distinct actions in a row and to then look at the information the Profiler has collected meanwhile: you create a new blog entry, look at the entry, and write a comment. After these operations, you open a view in which the Profiler tool shows those methods that took the most execution time.



On the first look, everything seems right. You are wondering however why the database calls (`Criteria.list`) from the method `findEntriesBetweenDates` took to long. You know that this method is called from the calendar view on the right hand side of the blog page.

Repeat Action

To verify your assumption, you reload the main blog page a couple of times by constantly pressing F5 in the browser. The rendering of a blog page is not affected by any data caching so that each call should cause the same number of database calls. Then, you have another look at the same view of the Profiler tool.



You notice that the amount of time spend in `Criteria.list` has relatively increased a lot. You also notice that for each page request, this method is called once from `fetchEntries` but much more often from `findEntriesBetweenDates`. This does not seem to be right.

You start your IDE, open the respective source file, navigate to the method `findEntriesBetweenDates`, and immediately find the following code:

```
List entryList = entryCriteria.list();

DatabaseEntry[] entries = (DatabaseEntry[]) entryList.toArray(new DatabaseEntry[entryList.size()]);
for (int i = 0; i < entries.length; i++) {
  Criteria categoryCriteria = session.createCriteria(Category.class);
  categoryCriteria.add(Restrictions.eq("name", entries[i].getCategory()));
  List categoryList = categoryCriteria.list();
  // TODO: Visiting categories probably not needed
}
```

Suddenly you realize that you began to implement a feature for which you needed to visit the category objects of all blog entries. You did not finish developing this function but the code that you left causes another database round trip for each blog entry. Besides the fact that this code is written very inefficiently, it does not even make any sense right now.

So you remove the code and take another look at the performance measures from your Profiler tool. Now that everything seems right you publish the new version of the application.

Acknowledgements

I would like to thank Sachin Bammi who gave important feedback as my shepherd for the EuroPLoP 2008 conference. I'd also like the thank the participants of the EuroPLoP 2008 workshop for their well thought-out and constructive comments and suggestions. In particular the section with unfinished patterns is based on their input. I hope to write a follow-up with the new material in more detail soon.

Resources

As far as the author is aware of there is no related work in pattern form that explains how to profile a software system. This section therefore presents references to other resources that explain the usage of Profilers in a more general form.

[1] List of profiling tools for Java:
http://www.javaperformancetuning.com/resources.shtml

[2] Jim Patrack, Handling memory leaks in Java programs:
http://www.ibm.com/developerworks/java/library/j-leaks/

[3] Brian Goetz, Java theory and Practice: Plugging memory leaks with weak references,
http://www.ibm.com/developerworks/java/library/j-jtp11225/index.html

[4] Tess Ferrandez: If broken it is, fix it you should. A blog about debugging and profiling .NET applications, http://blogs.msdn.com/tess/default.aspx

Patterns for Robust and Flexible Multimodal Interaction

Andreas Ratzka
Institute for Media, Information and Cultural Studies
University of Regensburg
D-93040 Regensburg, Germany
Andreas.Ratzka@sprachlit.uni-regensburg.de

Abstract

Multimodal interaction aims at more flexible, more robust, more efficient and more natural interaction than can be achieved with traditional unimodal interactive systems. In order to achieve this, the developer needs some design support in order to select appropriate modalities, to find appropriate modality combinations and to implement promising modality adaptation strategies. This paper presents a first sketch of an emerging pattern language for multimodal interaction and focusses on patterns for "flexible interaction" and patterns for "robust interaction". This work is part of a thesis project on pattern-based usability engineering for multimodal interaction.

1 Introduction

Multimodal interaction means interaction via several interaction channels such as speech, pointing device, graphics and the like. An interaction modality is defined by its interaction channel (acoustic, visual, haptic/tactile) and the interaction "language" (pointing, naming, emulating). Example modalities are pointing gestures, keyboard input, speech input, speech output, graphic output, and tactile output (e.g. vibrating steering wheels).

Although every interactive system combines at least two interaction modalities (one for input, another one for output) not everyone is necessarily multimodal. Multimodality implies the use of at least either more than one input channel or more than one ouptut channel. Although mouse-based pointing and keyboard input are different modalities, typical WIMP[1] and desktop applications are not classified as multimodal systems. Only when different *channels* (acoustic, visual, haptic/tactile) are used for either input or output, a system can be called multimodal.

According to Oviatt & Kuhn (1998) goals of multimodal interaction are

- flexibility and adaptability of the system with respect to users and context of use,
- high interaction robustness due to mutual disambiguation of input sources,
- interaction efficiency because of better integration into the work situation, and

[1] Windows, Icons, Menus, Pointing device

- the possibility to interact with the system in a natural way.

This work is related to a thesis project about usability engineering of multimodal interactive systems. One assumption of this thesis project is that, although multimodal interaction is a relatively new field with very little market penetration, there exists already a corpus of well founded research results and successful system implementations in which recurring patterns can be found (Ratzka & Wolff 2006).

The usability engineering lifecycle (Mayhew 1999) comprises the phases of requirements analysis, work reengineering, design standards, detailed design and implementation. All of these lifecycle steps need both specification formalisms and knowledge-based design support. Traditional approaches for multimodal interaction design provide implementation frameworks (Niedermaier 2003) or formalisms for specifying detailed design (Dragičević 2004, Duarte & Carriço 2006). Other work provides knowledge-based design support in the phases of requirement analysis and work reengineering (Bernsen 1999, Bürgy 2002, Obrenović et al. 2007).

The application of patterns can complement these approaches since they seem to be an appropriate tool for providing knowledge-based design support accross all lifecycle phases.

This pattern collection is based on a thorough literature review of multimodal interaction in industrial and research projects. The following questions helped to find an adequate categorisation of question-solution pairs and thus a basis for pattern mining:

- When should certain interaction modalities (speech input or pointing input, speech output or graphic output) be used?
- How should multiple interaction modalities (speech and pointing, speech and graphics etc.) be combined?
- How can modalities be adapted according to context of use (user, environment, or situation)?

2 Pattern Overview

This paper focusses on usability design patterns for robust and accessible multimodal interaction. Two pattern sub-collections are presented. The first one, *Flexible Interaction*, focuses on accessibility and cross-context usability. The abstract principle lying behind this sub-collection is giving the user the possibility to select appropriate interaction modalities according to context factors. In order to achieve this goal, this pattern group has to make use of suitable adaptation strategies which are described in the patterns *Multiple Ways of Input*, *Global Channel Configuration* and *Context Adaptation*.

The second sub-collection *Robust Interaction* focusses on avoiding and corroborating recognition errors as well as assuring communication between user and system. It contains the two general patterns *Redundant Input* and *Redundant Output* which present its abstract principle consisting in exploiting several interaction channels. The pattern *Important Message*, which has been described by Tidwell (1999, view section 5 in this paper) can be seen as concretisation of *Redundant Output*. The patterns *Multimodal N-best Selection* and *Spelling-based Hypothesis Reduction* are specializations of the pattern *Redundant Input*. They can be used in connection with the pattern *Speech-enabled Form* (Ratzka 2008, view section 5 in this paper), a specialization of the pattern *Form* (Tidwell 1999) which can be itself enriched with the patterns *Dropdown Chooser* and *Autocompletion* (Tidwell 2005). The patterns *Dropdown Chooser* (Tidwell 2005)

and *Continuous Filter* (van Welie & Trætteberg 2000) are used to implement *Multimodal N-best Selection* and *Spelling-based Hypothesis Reduction* respectively.

A short outline of patterns referenced in this paper but described elsewhere can be found in section 5.

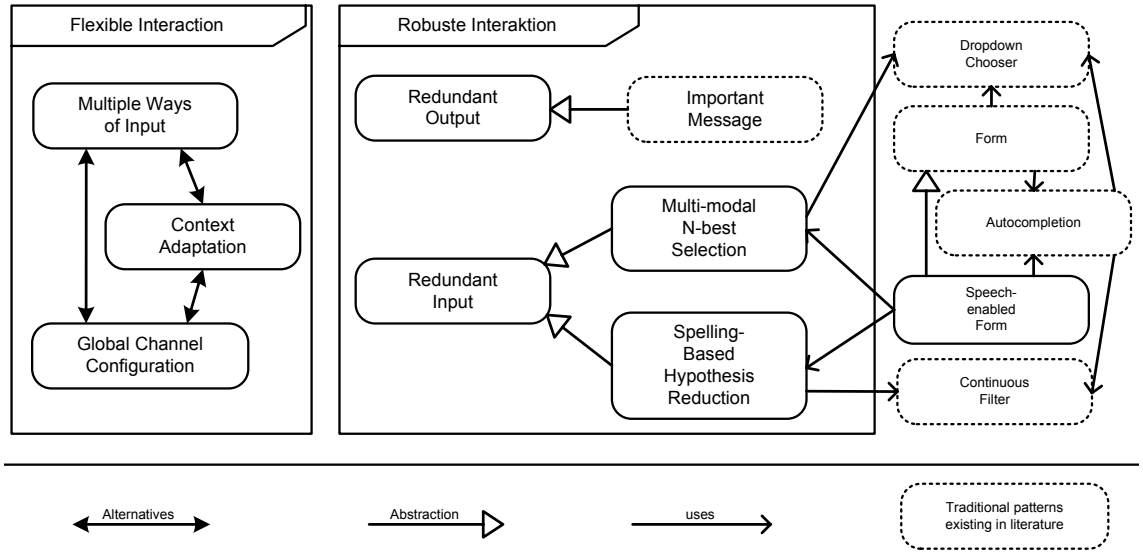


Figure 1: Pattern Map

3 Patterns for Flexible Multimodal Interaction

The patterns for flexible interaction focus on accessibility and usability across changing situations, environmental and other context factors. The three patterns described here provide each one different runtime modality adaptation strategies, i.e. ways of dynamically allocating interaction modalities:

- *Multiple Ways of Input* offers the user several alternative interaction techniques. The user is free to select the interaction modality that is most appropriate in the current context. He need not perform additional configuration steps.
- *Global Channel Configuration* provides several interaction profiles with each different configurations of input and output channels. This way, the system can be tailored to typical contexts of use. The user can switch the interaction profile in just one interaction step using an always-on-top widget or function button. Thus, the user can switch on/off audio output or speech recognition when appropriate.
- *Context Adaptation* requires the system to evaluate interaction history and environment data to adapt system behaviour accordingly. In contrast to *Multiple Ways of Input* and *Global Channel Configuration*, *Context Adaptation* provides system-initiated adaptation strategies.

These patterns try to resolve following forces:

- Typing is powerful for a lot of tasks but if the target user group includes typing-unskilled or even illiterate people other alternatives have to be used.
- Speech input is well suited for both text input and item selection. But in loud environments, speech recognition will be very error prone because the background noise masks the proper signal or is interpreted as input where there is none.
- Mechanical input via keyboards or pointing devices is widely applicable. But if the user's hands are occupied, wet, or dirty, there is no way to operate the systems.
- Graphic output and feedback is useful in a lot of situations but cannot be perceived in bad lighting conditions or by blind people. At the same time, speech output might be difficult to understand or even be overheard in loud environments.
- Speech output and input can make mobile interaction more comfortable. But confidential data must not be read out loudly in public environments.
- Environmental factors can be controlled via special installations such as specially mounted lamps, phone booths, directional speakers, earphones, or view shields, but these measures are not viable in every case such as mobile interaction.

Multiple Ways of Input

Context Context factors are not always predictable. This holds especially for mobile interaction in changing environments or public information kiosks that should be usable for quite different user groups.

Problem How can input modalities be adapted to the context of use without burdening the user with additional configuration tasks?

Forces

- The user should be able to interact with the system using preferred and task appropriate interaction styles. However, disabilities, or changing environmental conditions such as lighting or background noise may affect the usability and robustness of task-optimized interaction modalities.
- If background noise is low, speech input and output provide a valid interaction style. However, in public environments, bystanders might feel annoyed by persons conversing with an interactive system.
- Environmental factors can be controlled via special installations:
  - Specially mounted lamps help to overcome bad lighting conditions.
  - Phone booths reduce background noise and help to assure privacy.
  - Earphones and directional speakers avoid to annoy bystanders and enhance privacy.
  - View shields help to assure privacy.

  But these measures are not viable in every case such as mobile interaction.
- Users might differ according to preferences, language, reading, typing skills etc. The system can provide alternative interaction styles and adapt to the user. However, the system is not able to predict precisely which input modalities are most likely to be selected by the user in the current situation.

Solution Enable each system function to be operated via several alternative interaction modalities differing in the interaction channel.

Enable the user to select his preferred interaction channel, be it speech, typing or pointing. This modality should be active, i.e. the user should not have to switch the system configuration to use it.

The system has to be designed in a way, that the user can change interaction modalities wherever sensible, even for single interaction steps while performing a certain task.

Labels, help messages, prompts, console and speech commands should share a uniform vocabulary in order to minimise habituation and learning efforts.

Consequences

- Providing several alternative interaction styles, the system supports access for physically disabled or illiterate people.
- As the user can select from a set of complementary modalities which are each differently affected by environmental factors, the system can be used in varying environmental conditions.
  - In loud or public environments the users can simply sidestep to pointing / text input.
  - When background noise is low users can opt for speech interaction.
- Expert users can estimate whether speech input or pointing gestures are possible or desirable at the current moment.
- Interaction can be assured even though no special installations such as view shields or directional speakers are available.
- It is up to the users to select their preferred interaction style. They do not need to rely on mystic autoadaptation mechanisms. Instead, they gain self-confidence as they are controlling the system which leads to higher user satisfaction.
- First time users might not know which interaction styles are available at all. The system must provide effective help and prompting strategies that reveal alternative interaction modalities.
- The more flexible a system is, the more planning, testing and reviewing is needed during design as the number of error sources increases rapidly with system complexity.

Rationale According to user characteristics, preferences, environment and situation, different interaction modalities are preferable (Oviatt et al. 2000, Bürgy 2002, Obrenović et al. 2007).

Users can judge better than the system, which interaction modality and style is appropriate, e.g. whether background noise or bad lighting impedes the use of speech or graphics, whether surrounding people might feel annoyed because of spoken interaction or whether private information is being conveyed.

Ibrahim & Johansson (2002) have shown for their multimodal TV guide, that users prefer, when they can choose to use direct manipulation and speech either unaccompanied or combined in order to adapt to the current context of use.

Users learn to estimate and select the most context-appropriate modality. After recognition errors, users tend to switch the interaction modality (Oviatt et al. 2000, Oviatt & vanGent 1996, Oviatt et al. 1999, Oviatt 1999).

Known Uses In the SmartKom system (Reithinger et al. 2003) the user can interact either via pen or speech.

Mobile systems such as Microsoft's MiPad (Miyazaki 2002, Microsoft) and IBM's Personal Speech Assistant (Comerford et al. 2001) are good examples of systems allowing users to flexibly select input modalities. The same holds for driver assitance system such as the ones decribed in Neuss (2001) and Pieraccini et al. (2004).

Further examples of this pattern can be found in the interactive TV guide by Ibrahim & Johansson (2002), and MICASSEM (McCaffery et al. 1998).

Related Patterns This pattern is on the one hand an alternative to *Global Channel Configuration* and *Context Adaptation*. On the other hand these patterns can be combined.

Whereas *Global Channel Configuration* requires an additional interaction step to select the input / output profile of the system, *Multiple Ways of Input* offers a wider spectrum of input modalities which can be used without additional configuration steps. In contrast to *Global Channel Configuration*, this pattern is restricted to the adaptation of user input.

That's why this pattern should be used with *Context Adaptation*. This way, system output can be adapted according to the user's input behaviour.

In contrast to system-driven *Context Adaptation*, *Multiple Ways of Input* offers, similarly to *Global Channel Configuration*, user-driven system adaptation.

Global Channel Configuration

Context Interactive devices that offer several alternative and complementary interaction channels such as audio input and output, typing and graphic manipulation have to be adapted to the context of use.

Problem How can interaction (input and output) be adapted to the current context of use, while giving the user control over the system without burdening him with too much configuration tasks?

Forces

- One can keep several input channels active and leave it up to the user to select the most appropriate interaction modality for the current context. However, the system might try to interpret input from unused distorted channels, e.g. because of background noise. This might lead to *false positives*, that is the system misinterprets background noise as input.
- Redundant output via several channels can assure information perception by users. However, in public environments, bystanders might feel annoyed by persons conversing with an interactive system. In the same way, it is not desirable, that private data be read out loudly by the system.
- The system may analyse interaction behaviour, lighting conditions, background noise, movement and position changes and adapt to the user and context of use. However, the system does not find out as fast as the user does which interaction modalities are most appropriate in the current context of use or for the current user.
- Even if automatic adaption works quite well, many users will prefer being able to control the system.

Solution Provide several interaction profiles with input and output channel configurations tailored to each context of use. Enable the user to select the interaction profile with only one additional interaction step.

Display for instance always on top buttons with self explaining icons or provide physical *push-to-talk* and *mute* buttons etc. so that the user can select the interaction profile (speech input, audio output), the notification profile of a mobile phone (ringing, vibrating, mute) with one click.

Consequences

- The user can quickly react to context changes and reconfigure the system accordingly in one interaction step:
  - Input channels such as speech input can be deactivated when necessary (in loud environments) without great effort by simply clicking the necessary button.
  - In order not to disturb bystanders in public environments, the user can deactivate speech output with one click.

- Users feel better when exercising control over the system instead of being delivered to its "caprices".
- Users have to do at least one additional interaction step which might be annoying anyway.
- Users might by fault select an inappropriate interaction profile.
- The users might not be able to rembember the current configuration state of the system and that they have to reconfigure the system at each situation change.

Rationale According to user preferences, abilities, environment and situation, different interaction modalites are appropriate and have to be preferred (Oviatt et al. 2000, Bürgy 2002, Obrenović et al. 2007).

Users learn to estimate and select the most context-appropriate modality. After recognition errors, users tend to switch the interaction modality (Oviatt et al. 2000, Oviatt & vanGent 1996, Oviatt et al. 1999, Oviatt 1999).

Known Uses The multimodal map-based system *SmartKom Mobile* covers the use cases of pedestrian and automotive navigation as well as map-based queries (Malaka et al. 2004) and allows the user to switch between several interaction modes (cf. Wasinger 2006, p. 59):

- *Default*: All input and output modalities are supported.
- *Listener*: Speech+graphics are supported for output, for input only pen gestures are possible.
- *Silent*: Only graphics for output and pen gestures for input are supported.
- *Speech Only*: Only speech interaction is active.

Some mobile phones such as *Nokia E71* offer the user to select profiles such as *office* or *home*. In addition to different startup screens, these profiles can be set to an appropriate context-dependent notification mode (ringing, vibrating).

With some restrictions, desktop applications, operating system environments, and multimedia applications that provide an audio icon in the system tray for setting the system's audio characteristics can be seen as examples for this pattern.

Push-to-talk buttons in some speech based driver assistance systems are examples, too: When operated once, speech output is stopped and speech input activated. Operated once more, speech interaction can be deactivated and, lateron, reactivated.

Related Patterns This pattern is an alternative to *Multiple Ways of Input* and *Context Adaptation* but can be used in combination with them, too.

In contrast to *Multiple Ways of Input*, which offers the user to select among several alternative input modalities without having to perform additional configuration actions, this pattern requires one additional interaction step. Furthermore, it comprises, in contrast to *Multiple Ways of Input*, the adaptation of system output.

In contrast to system-driven *Context Adaptation*, *Global Channel Configuration* and *Multiple Ways of Input* are forms of user-initiated system adaptation.

Context Adaptation

Context Examples for this pattern can be found in interactive devices that support different alternative and complementary interaction channels such as audio output and input, typing and graphic manipulation.

Problem How can interaction (input and output) be adapted to the current situation, environment and user without the user having to perform additional interaction steps?

Forces

- Redundant output via several channels can assure information perception by users. However, superfluous spoken output might disrupt or slow down the user's secondary task or annoy third persons.
- One can keep several input channels active and leave it up to the user to select the most appropriate interaction modality for the current context. However, the system might try to interpret input from unused distorted channels. This might lead to *false positives*, that is the system misinterprets background noise as input.
- Alternatively, one could let the user configure system input and output according to the current context of use. But, as long as sufficient information is available to the system additional configuration steps should be avoided.
- Letting the user configure the system himself seems not to be a problem at first. But the user might not be able to rembember the current configuration state of the system and to reconfigure the system at each situation change.

Solution The system should analyse as much assured context information as available to setup system configuration autonomously.

One information source that can be used is the interaction history:

- If the user interacts via speech, it is not clear where the user looks. In this case speech output should complement display updates.
- If the user does not want to annoy surrounding people, he will avoid speech interaction.
- If the user interacts with pointing device or keyboard – might be in order to avoid annoying surrounding people – he is usually looking at the display such that speech output is superfluous.

Other aspects of system state can be exploited for adaptation, too: A driver assistance system should disable touch screen input while the car is driving – the driver should keep his hands on the steering wheel. Smartphones should disable touch screen input while the user is making a telephone call – the ear of the user should not lead to unwanted actions.

Consequences

- Information output can be restricted according to the context of use.
- False positive recognitions can be avoided via context dependent recogniser activation.
- The user does not need to reconfigure the system repeatedly where this can be done by the system itself.
- The user need not remember configuration states and reconfigure the system at each situation change.
- Automatic adaptation can fail or be unappropriate. That's why the user should always have the possibility to carry over control and perform interaction configuration himself.

Known Uses In the SmartKom system (Reithinger et al. 2003) the user can interact either via pen or speech. System feedback is adapted in a way that fits to the user's attention: The TV-guide subsystem of SmartKom presents the TV-program usually as a listing but reads out spoken feedback to spoken queries. When the user is watching TV, the system presents the program list verbally, too, because it supposes that the visual channel is occupied.

Driver assistance system such as the ones decribed in Neuss (2001) and Pieraccini et al. (2004) offer the user to interact using speech or manual input devices. System output is adapted acordingly.

Smartphones disable touch screen input during telephone calls.

Related Patterns This pattern is an alternative to *Global Channel Configuration* and *Multiple Ways of Input* but may be used complementarily to these ones. In contrast to these user-driven adaptation strategies, *Context Adaptation* implements a system-initiated adaptation strategy.

This pattern can be used as complement to *Multiple Ways of Input* in order to update system output according to user input.

In addition to *Context Adaptation*, *Global Channel Configuration* should be used, to give the user control over the system.

4 Patterns for Robust Multimodal Interaction

This subcollection comprises a set of four patterns: two abstract two concrete. Both abstract patterns *Redundant Output* and *Redundant Input* go back to the same basic principle of mutual disambiguation of redundant signals. Whereas the pattern *Redundant Output* is used to present redundant data to the user such that the user is more likely to understand or at least perceive the information conveyed by the system, the pattern *Redundant Input* fuses user input coming from several channels in order to reduce recognition and interpretation errors.

The pattern *Redundant Output* has no concretisation within this collection. Tidwell's pattern *Important Message* (Tidwell 1999, view section 5 in this paper) can be seen as a concretisation of this one, as it suggests the usage of several modalities (visual and auditive signals) to attract the user's attention and convey urgent information.

For the pattern *Redundant Input* two refining patterns are given:

- *Multimodal N-best Selection* combines several interaction modalities for input and disambiguation dialogs. The user first speaks a word or command. As speech recognition is not a sharp but rather statistical process a list of *n* best hypotheses is returned. The user can now directly select from this list e.g. via pointing. When this step has to be done via speech, too, alternatives to simply re-speaking the item – e.g. naming the row number – should be supported and prompted for in order to avoid endless loops of repeated errors due to intrinsic acoustic confusability.

- *Spelling-based Hypothesis Reduction* combines several input modalities to reduce recognition errors. Typically, the user first inputs some letters via typing in order for the system to reduce the set of possible alternatives. Then, the user speaks the desired entry which can be properly recognised by the system. Spelling can be done after spoken selection, as well. Then, the system has to recalculate the recognition hypotheses according to the updated list of available alternatives.

These pattern, although focusing on robust interaction, are closely linked to patterns for efficient multimodal interaction, especially the patterns *Voice-based Interaction Shortcut* and *Speech-enabled Form* (Ratzka 2008, view section 5 in this paper) as well as to traditional user interface patterns such as *Continuous Filter* (van Welie & Trætteberg 2000, cf. section 5), *Form* (Tidwell 1999, cf. section 5), *Autocompletion* (Tidwell 2005, cf. section 5) and *Drop-down Chooser* (Tidwell 2005, cf. section 5).

Redundant Output

Context Communication channels might be unpredictably distorted due to bad lighting conditions, background noise, technical (network) problems or disabilities such as speech, motor or perception disorders.

Public systems that should be accessible for everybody should use this pattern especially during the first interaction steps when it is not clear which interaction channels are appropriate for the current user.

Problem How to assure information output when communication channels are distorted in an unforseeable way?

Forces

- The system can be configured or adapted to output information using modalities that are less affected by channel disorders. However, in some cases several interaction channels are distorted to some degree. Examples are:
  - Visually impaired people or illiterate people that want to interact in loud environments.
  - Deaf people that want to interact in bad lighting conditions or while moving around.
- Potential channel distortions might be circumvented by selecting alternative interaction channels. However, if the potentially distorted channel were otherwise the best candidate, abandoning this channel cannot be justified.
- The system can use those modalities that are most appropriate in the current environmental context. However, when the user's attention does not fit to the situation he might miss important notifications. E.g. when the system uses purely visual output due to high background noise level but the user's visual attention is focused elsewhere the system fails to notify the user.

Solution Combine several output channels in order to make use of redundancy. Information should be output both visually and acoustically and possibly even in a haptic way (e.g. using vibration) to raise the probability that it is perceived and can be understood by the user.

Consequences

- The use of several channels raises the probability that the user is able to perceive the information conveyed to him by the system. Visually impaired people in loud environments or deaf people in bad lighting conditions can process more data when output is presented redundantly to them.
- You don't need to abandon an output channel totally, only because it might be distorted. Visually impaired people might have problems reading a text and recognise each letter. After hearing the spoken variant and knowing what the text is about the visual representation can be used as memory hook. The same might be true for dark environments or mobile scenarios, when it is difficult to fix visual attention to the text.

- It is more likely to attract the user's attention when information is output via several channels, e.g. both audio and sight, than when only one output modality is used.

Rationale Independent disturbances of different channels rarely affect the same aspects of the content.

Multi-channel feedback of written and spoken text has proven to be effective for elderly (Emery et al. 2003) and visually impaired users, especially those suffering from *AMD*[2] (Vitense et al. 2002, Jacko et al. 2003, 2004, Edwards et al. 2004).

In the context of language understanding, it has been proven that users understand language better when they can read the lips of their interlocutor simultaneously to hearing (Sumby & Pollack 1954, Neely 1956, Binnie et al. 1974, Erber 1969, 1975, Summerfield 1979, Schomaker et al. 1995). This holds especially in loud environments.

Plosives ([p], [t], [k], [b], [d], [g]) sound similar and are likely to be confused when sound quality is low. At the same time these phones have distinctive lip shapes such as open lips (in the case of [g] and [k]) vs. initially closed lips (in the case of [b] and [p]). Lip shapes may differ for some similar sounding vowels, too.

Known Uses Some interactive systems such as *PPP* (André et al. 1996), *NUMACK* (Kopp et al. 2004), *COMIC* (Foster et al. 2005) and *SmartKom* ("Smartakus" Wahlster et al. 2001) display talking heads, in order to exploit the advantages of audiovisual language understanding (Benoît et al. 1998).

Mobile phones combine visual (blinking), auditive (ringing) and haptic (vibrating) signals in order to notify the user about phone calls or incoming short text messages.

Related Patterns Tidwell's *Important Message* (Tidwell 1999) is a concretisation of this pattern. In this case, information is output multimodally in order to attract the user's attention. Beyond attention attraction and raising perception probability, the generic pattern *Redundant Output* tries to overcome comprehension problems caused by context factors such as bad light or background noise.

[2] Age-related Macular Degeneration

Redundant Input

Context Communication channels might be unpredictably distorted due to bad lighting conditions, background noise, technical (network) problems or disabilities such as speech, motor or perception disorders.

Problem How to assure input when communication channels are distorted in an unforseeable way?

Forces

- The system can be configured or adapted to recognise and interpret that modality that is less affected by channel disorders, but in some cases all available interaction channels are distorted to some degree.
  - In loud environments users with motor disabilities or illiterate people have problems to interact with the system.
  - In dark environments or in hands-free scenarios (e.g. while carrying a bag, wandering around, driving a car) people with speech disorders have problems to input data.
  - In exerted conditions, both speech input and pen gestures are problematic.
- Interaction can be alleviated if passive modalities for data input (gaze input, free gestures) or authentication (voice recognition, face recognition) are used. However, these interaction channels are error prone so that they cannot be applied directly.

Solution Combine several interaction channels in order to make use of redundancy. Input coming from several channels (visual: e.g. lip movements, auditive: e.g. speech signal) should be interpreted in combination in order to reduce liability to errors.

Consequences

- The use of several channels raises the probability that the system is able to recognise and interpret the information input by the user in the desired way.
- Even if several channels are distorted the distortion rarely affects exactly the same pieces of information. Fusion mechanisms allow for reconstructing of at least some part of the input information.
- "Imperfect", error prone interaction channels can be combined to mutually disambiguate recognition errors.

Rationale Independent disturbances of different channels rarely affect the same aspects of the content. That's why for instance audio-visual speech recognition, which combines acoustic signals and lip movement analysis, leads to better recognition performance than unimodal speech recognition (cf. Benoît et al. 1998, S. 24 f.):

Plosives ([p], [t], [k], [b], [d], [g]) sound similar and are likely to be confused when sound quality is low. At the same time these phones have distinctive lip shapes such as open lips (in

the case of [g] and [k]) vs. initially closed lips (in the case of [b] and [p]). Lip shapes may differ for some similar sounding vowels, too.

Channel distortions rarely affect both the recognition of a specific phoneme in the acoustic signal and of the corresponding viseme in the visual signal in the same way. Fusion algorithms allow to combine sound pieces of information from several channels such that some distorted parts can be reconstructed.

Studies conducted by (Oviatt 1999) revealed that an appropriate recogniser architecture that combines gesture and speech recognition can reduce recognition errors. This was shown for non-native speakers, in loud environments (Oviatt 2000a, b, c) and for exerted conditions (Kumar et al. 2004).

Known Uses This pattern is manifested in very different application areas including data input (audio-visual speech recognition), scene analysis (Wachsmuth 2001), person identification (Yang et al. 1998, 1999, Hazen et al. 2003, Jain 2003, Snelick et al. 2003), emotion recognition (Nasoz et al. 2002, Lisetti & Nasoz 2002, Busso et al. 2004, Gunes et al. 2004, Zeng et al. 2004) and the like. Following modality combinations are used:

- virtual reality and speech (Kaiser et al. 2003),
- gaze direction and speech (Zhang et al. 2004, Tan et al. 2003, Tanaka 1999, Campana et al. 2001),
- lip-reading in loud environments (Saenko et al. 2004), e.g. to filter out simultaneous speakers (Patterson & Gowdy 2003),
- speech and gesture (Holzapfel et al. 2004, Chai & Qu 2005),
- voice, ink and touchtone (Trabelsi et al. 2002),
- biometrics, voice and face to identify persons (Yang et al. 1999, Hazen et al. 2003).

Related Patterns *Spelling-based Hypothesis Reduction* as well as *Multimodal N-best Selection* are refinements of this pattern.

Multimodal N-best Selection

Context This pattern can be used in multimodal systems that offer speech input of unconstrained text or speech-based selection of items from very large sets such as timetable or navigation systems.

Problem Speech recognition is a statistical process. The recognition of input phrases results in a set of several recognition hypotheses. It is frequently the case that the original input phrase is included in the n-best set but does not coincide with the system's best estimate.

Forces

- When a speech input attempt fails the user can be prompted to repeat or to switch to another interaction modality. But it is inefficient to throw away input data which has failed the goal just by a hair.
- Playing back the *n* best recognition hypotheses, prompting for (spoken) selection and reducing the speech recognition vocabulary to this reduced list of *n* items can correct the error in just one further interaction step. However, items contained in the n-best list are likely to have some acoustic similarity so that they might be mixed up repeatedly by the recogniser.
- Playing back just the item on top of the *n* best recognition hypotheses and prompting for accepting or rejecting may resolve this problem in a few steps, but if the desired item is only the fifth (sixth, seventh ...) best recognition hypothesis five (six, seven ...) error corroboration steps are needed.

Solution Provide the user a means of selecting the correct result from a set of recognition hypotheses via pointing or key presses.

In order to satisfy cases where the desired item cannot be found in the n-best list there has to be a way of explicitly leaving the list selection dialog and to start over the input attempt.

Consequences

- Imperfect recognition results are not thrown away but reused in subsequent interaction steps.
- Recurring recognition problems due to confusability are avoided through alternative selection techniques. Instead of re-speaking the misrecognised item, the user can point to the item displayed in the list.
- Frequently, instead of endless error correction loops, this pattern helps to correct recognition errors in just one additional interaction step.

Rationale Suhm et al. (2001) point out that re-speaking the same word or phrase after recognition failure, although natural, is not the most promising form of error recovery in interactive systems. Changing the input modality to list selection seems to be more promising and has been suggested by Ainsworth & Pratt (1992) and Murray et al. (1993).

Known Uses Directory assistance, timetable information systems, speech-based driver assistance systems support n-best selection.

Related Patterns This pattern can be used in conjunction with *Speech-enabled Form*, *Drop-down Chooser* (Tidwell 2005) and *Autocompletion* (Tidwell 2005) to alleviate error handling in speech-enhanced input forms.

This pattern and *Spelling-based Hypothesis Reduction* are refinements of *Redundant Input*. *Multimodal N-best Selection* makes use of *Drop-down Chooser* (Tidwell 2005) to provide the user a non-speech alternative for selecting from a list of the *n* most likely recognition hypotheses and to avoid repeated errors.

Variant The solution described in this pattern is not restricted to strictly multimodal interaction. Speech-only systems provide similar speech-based approaches of selecting the desired item from a list of hypotheses.

If n-best selection via speech is supported, it is important to offer input modifications to avoid repeated errors. The user should be given the possibility to select the desired option via speaking the line number or re-speaking the item in combination with some distinctive attribute such as the first letter(s).

In these cases, the user has to be prompted in a way that reveals alternative selection strategies apart from simple re-speaking, e.g.: "Did you mean *one* Jonathan Smith, *two* John Griffith, *three* Joseph Reddish or *new input*".

While the list is being read out the user should have the possibility to interrupt the playback. In full-duplex systems with cleanly separated channels for audio input and output barge-in can be used. That means that the system playback is stopped when the user starts speaking. In half-duplex systems the user should be able to interrupt system playback using a push-to-talk button. In the cases of both barge-in and push-to-talk the interruption moment can be used as a further information source to estimate the selected item (Balentine 1999, Balentine & Morgan 2001).

Spelling-based Hypothesis Reduction

Context Examples where this pattern can be used are systems that offer speech input of unconstrained text or speech-based selection of items from very large sets such as timetable or navigation systems. Errors are particularly likely in cases, when the user has to select from lists with similar sounding words or words with inconsistent pronunciation such as foreign names.

Problem Large recognition vocabularies entail error-prone recognition, especially when many similar sounding words have to be distinguished.

Forces

- Speech input can be used for selecting items from a list that cannot be displayed completely on a small screen, but if the list is too large for speech recognition or includes several similar sounding or problematic words, speech recognition is likely to fail. Problem areas are names in directory assistance systems as well as album and song titles in entertainment systems.
- In the case of recognition failure, the user can switch to text input (typing), but in some applications such as driver assistance systems only unhandy (if any) string input facilities are available.
- Typing the first letters can reduce the size of the selection list so that pointing is possible again, but in some applications such as navigation systems there might be a lot of characters to be input (using an impractical input device) before the list is reduced to a displayable size.
- Some speech recognisers, especially those for small devices, have only restricted resources such that only small vocabularies e.g. for number recognition or for recognising letters of the alphabet are supported. But operating applications this way only provides little added value in contrast to using a small keypad, especially since letters some of which sound similar are likely to be confused by the recogniser.

Solution Offer the user to input the first letters of the input tokens via typing. Use this substring to reduce the size of selectable items (i.e. of the speech recognition vocabulary) to head-matching strings. Using this vocabulary forthcoming speech inputs can be recognised more robustly.

Record speech input attempts and keep this recording available for some forthcoming interaction steps. Failed speech input can thus be reinterpreted afterwards successfully with reduced vocabulary.

Consequences

- By inputting quite few letters, the set of alternatives can be reduced to a size that – although still unsuited for pointing-based list selection – allows robust speech recognition.
- The user needs only to input few letters. This is important in cases where text input is inconvenient.

- There is no need to navigate and scroll through lists.
- Recognition of names, song titles and other problematic words becomes more robust.
- This technique of reducing speech recognition vocabularies simplifies speech recognition on platforms with limited resources.

Rationale The reduction of speech recognition vocabulary can improve recognition performance significantly.

Known Uses Marx & Schmandt (1994) describe the messaging system *Chatter* which allows the user to input contact names via voice spelling, touch-tone spelling and speech-based naming in a combined fashion.

The prototype of a multimodal driver assistance system by Neuss (2001) allows the user to input the first letters using a rotary knob mounted on the centre console, so that the speech recognition vocabulary can be reduced.

Other examples that go in the same direction can be found in Suhm et al. (2001) and Tan et al. (2003).

Related Patterns This pattern can be used in conjunction with *Speech-enabled Form*, *Drop-down Chooser* (Tidwell 2005) and *Autocompletion* (Tidwell 2005) to alleviate error handling in speech-enabled input forms.

This pattern and *Multimodal N-best Selection* are refinements of *Redundant Input*. *Spelling-based Hypothesis Reduction* uses (or is) some kind of variant of *Continuous Filter* (van Welie & Trætteberg 2000). Instead of filtering the items of a selection list, the recognition vocabulary is reduced according to the letters input by the user.

Variant Some systems allow the user to dictate characters (voice spelling) in a purely speech-based way to reduce the recognition vocabulary.

In this case, phonetic alphabets can be used to reduce recognition failures. But this is only feasible if the target user group is expected to be proficient in that phonetic alphabet.

When a phonetic alphabet is supported the user is not proficient in, this might result in spontaneous wordings such as *Motel* instead of *Mike* or *October* instead of *Oscar*. This way, recognition errors might even increase.

5 Related Patterns from other Collections

5.1 Multimodal User Interface Patterns

This is only a fraction of the patterns identified during this work. Other patterns focus on fast / efficient interaction. The following table outlines only the pattern *Speech-enabled Form* (cf. Ratzka 2008), since it is referenced several times in this paper:

Name	Problem	Solution
Speech-enabled Form	How to simplify string input in form filling applications?	Let the user select the desired form field via pointing and input the desired value via speech.

5.2 Related Patterns from other Authors

These patterns for multi-modal interaction are closely related to traditional user interface patterns described by other authors:

Name Reference	**Problem**	**Solution**
Continuous Filter (van Welie & Trætteberg 2000)	"The user needs to find an item in an ordered set".	"Provide a filter component with which the user can in real time filter only the items in the data that are of interest."
Important Message (Tidwell 1999)	"How should the artefacts convey this information to the user?".	"Interrupt whatever the user is doing with the message, using both sight and sound if possible."
Form (Tidwell 1999, Sinnig et al. 2004), (www.welie.com)	"The user must provide structural textual information to the application. The data to be provided is logically related"	"Provide users with a form containing the necessary elements. Forms contain basically a set of input interaction elements and are a means of collecting information [...]".
Drop-down Chooser (Tidwell 2005)	"The user needs to supply input that is a choice from a set [...], a date or time, a number, or anything other than free text typed at a keyboard. [...]".	"For the Drop-down Chooser control's 'closed' state, show the current value of the control in either a button or a text field. To its right, put a down arrow. [...] A click on the arrow (or the whole control) brings up the chooser panel, and a second click closes it again [...]".
Autocompletion (Tidwell 2005)	"The user types something predictable, such as a URL, the user's own name or address, today's date, or a filename [...]".	"With each additional character that the user types, the software quietly forms a list of the possible completions to that partially entered string [...]".

6 Conclusion

This paper outlines an emerging pattern language for multimodal interaction which is far from being complete. Despite the research history of over twenty years, multimodality is still a research-centric field. It begins to reach some dissemination in the fields of automotive, industrial and mobile applications. That is why interaction design support is needed. Interaction design patterns constitute a challenging and exciting approach to this domain.

7 Acknowledgements

I would like to thank my shepherd Michael vanHilst and the members of the workshop group at EuroPLoP 2008 for the numerous suggestions and valuable feedback.

References

Ainsworth & Pratt 1992
AINSWORTH, W.A.; PRATT, S.R.: Feedback strategies for error correction in speech recognition systems. In: *Int J. Man-Mach. Stud.* 36 (1992), Jun., Nr. 6, S. 833–842

André et al. 1996
ANDRÉ, E.; MULLER, A. J.; RIST, T.: The PPP Persona: A Multipurpose Animated Presentation Agent. In: AL., Catarci et (ed.): *Advanced Visual Interfaces*, ACM Press, 1996, S. 245–247

Balentine 1999
BALENTINE, B.: Re-Engineering The Speech menu. In: GARDNER-BONNEAU, D. (ed.): *Human Factors and Voice Interacive Systems*. Norwell, Massachussetts: Kluwer Akademic Publishers, 1999, S. 205–235

Balentine & Morgan 2001
BALENTINE, Bruce; MORGAN, D. P.: *How to Build a Speech Recognition Application. A Style Guide for Telephony Dialogues*. EIG Press, 2001

Benoît et al. 1998
BENOÎT, C.; MARTIN, J.-C.; PELACHAUD, C.; SCHOMAKER, L.; SUHM, B.: Audio-Visual and Multimodal Speech Systems. Version: 1998. http://www.limsi.fr/Individu/martin/publications/download/chmultimodal.ps. In: GIBBON, D. (ed.): *Handbook of Standards and Resources for Spoken Language Systems - Supplement Volume*. 1998

Bernsen 1999
BERNSEN, Niels O.: *Multimodality in Language and Speech Systems - from theory to design support tool.* Lectures at the 7th European Summer School on Language and Speech Communication (ESSLSC). http://www.nis.sdu.dk/ nob/stockholm.zip. Version: Juli 1999, retrieved on: 20.06.2008

Binnie et al. 1974
BINNIE, C. A.; MONTGOMERY, A. A.; JACKSON, P. L.: Auditory and visual contributions to the perception of consonants. In: *Journal of Speech & Hearing Research* 17 (1974), S. 619–630

Bürgy 2002
BÜRGY, Christian: *An Interaction Constraints Model for Mobile and Wearable Computer-Aided Engineering Systems in Industrial Applications*, Department of Civil and Environmental Engineering, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, Diss., 2002

Busso et al. 2004
BUSSO, C.; DENG, Z.; YILDIRIM, S.; BULUT, M.; LEE, C.-M.; KAZEMZADEH, A.; LEE, S.; NEUMANN, U.; NARAYANAN, S.: Analysis of emotion recognition using facial expressions, speech and multimodal information. In: *ICMI '04: Proceedings of the 6th international conference on Multimodal interfaces*. New York, NY, USA: ACM Press, 2004, S. 205–211

Campana et al. 2001
CAMPANA, E.; BALDRIDGE, J.; DOWDING, J.; HOCKEY, B.-A.; REMINGTON, R.-W.; STONE, L.-S.: Using eye movements to determine referents in a spoken dialogue system. In: *PUI '01: Proceedings of the 2001 workshop on Perceptive user interfaces*. New York, NY, USA: ACM Press, 2001, S. 1–5

Chai & Qu 2005
CHAI, Joyce Y.; QU, Shaolin: A salience driven approach to robust input interpretation in multimodal conversational systems. In: *HLT '05: Proceedings of the conference on Human Language Technology and Empirical Methods in Natural Language Processing*. Morristown, NJ, USA: Association for Computational Linguistics, 2005, S. 217–224

Comerford et al. 2001
COMERFORD, L.; FRANK, D.; GOPALAKRISHNAN, P.; GOPINATH, R.; SEDIVY, J.: The IBM Personal Speech Assistant. In: *Proc. of IEEE ICASSP'01* DARPA, 2001, S. 319–321

Dragičević 2004
DRAGIČEVIĆ, Pierre: *Un modèle d'interaction en entrée pour des systèmes interactifs multi-dispositifs hautement configurables*, Université de Nantes, école doctorale sciences et technologies de l'information et des matérieaux, Diss., Mars 2004. http://www.dgp.toronto.edu/~dragice/these/html/memoire_dragicevic.html, retrieved on: 20.06.2008

Duarte & Carriço 2006
DUARTE, C.; CARRIÇO, L.: A conceptual framework for developing adaptive multimodal applications. In: *IUI '06: Proceedings of the 11th international conference on Intelligent user interfaces*. New York, NY, USA: ACM, 2006, S. 132–139

Edwards et al. 2004
EDWARDS, P. J.; BARNARD, L.; EMERY, V. K.; YI, J. S.; MOLONEY, K. P.; KONGNAKORN, T.; JACKO, J. A.; SAINFORT, F.; OLIVER, P. R.; PIZZIMENTI, J.; BADE, A.; FECHO, G.; SHALLO-HOFFMANN, J.: Strategic design for users with diabetic retinopathy: factors influencing performance in a menu-selection task. In: *Assets '04: Proceedings of the 6th international ACM SIGACCESS conference on Computers and accessibility*. New York, NY, USA: ACM Press, 2004, S. 118–125

Emery et al. 2003
EMERY, V. K.; EDWARDS, P. J.; JACKO, J. A.; MOLONEY, K. P.; BARNARD, L.; KONGNAKORN, T.; SAINFORT, F.; SCOTT, I. U.: Toward achieving universal usability for older adults through multimodal feedback. In: *CUU '03: Proceedings of the 2003 conference on Universal usability*. New York, NY, USA: ACM Press, 2003, S. 46–53

Erber 1969
ERBER, N. P.: Interaction of audition and vision in the recognition of oral speech stimuli. In: *Journal of Speech & Hearing Research* 12 (1969), S. 423–425

Erber 1975
ERBER, N. P.: Auditory-visual perception of speech. In: *Journal of Speech & Hearing Disorders* 40 (1975), S. 481–492

Foster et al. 2005
FOSTER, M. E.; WHITE, M.; SETZER, A.; CATIZONE, R.: Multimodal generation in the COMIC dialogue system. In: *ACL '05: Proceedings of the ACL 2005 on Interactive poster and demonstration sessions*. Morristown, NJ, USA: Association for Computational Linguistics, 2005, S. 45–48

Gunes et al. 2004
GUNES, Hatice; PICCARDI, Massimo; JAN, Tony: Face and body gesture recognition for a vision-based multimodal analyzer. In: *VIP '05: Proceedings of the Pan-Sydney area workshop on Visual information processing*. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2004, 19–28

Hazen et al. 2003
HAZEN, Timothy J.; WEINSTEIN, Eugene; PARK, Alex: Towards robust person recognition on handheld devices using face and speaker identification technologies. In: *ICMI '03: Proceedings of the 5th international conference on Multimodal interfaces*. New York, NY, USA: ACM Press, 2003, S. 289–292

Holzapfel et al. 2004
HOLZAPFEL, Hartwig; NICKEL, Kai; STIEFELHAGEN, Rainer: Implementation and evaluation of a constraint-based multimodal fusion system for speech and 3D pointing gestures. In: *ICMI '04: Proceedings of the 6th international conference on Multimodal interfaces*. New York, NY, USA: ACM Press, 2004, S. 175–182

Ibrahim & Johansson 2002
IBRAHIM, Aseel; JOHANSSON, Pontus: Multimodal dialogue systems: A case study for interactive tv. In: *Proceedings of 7th ERCIM Workshop on User Interfaces for All*. Chantilly, France, 2002, 209–218

Jacko et al. 2004
JACKO, J. A.; BARNARD, L.; KONGNAKORN, T.; MOLONEY, K. P.; EDWARDS, P. J.; EMERY, V. K.; SAINFORT, F.: Isolating the effects of visual impairment: exploring the effect of AMD on the utility of multimodal feedback. In: *CHI '04: Proceedings of the SIGCHI conference on Human factors in computing systems*. New York, NY, USA: ACM Press, 2004, S. 311–318

Jacko et al. 2003
JACKO, J. A.; SCOTT, I. U.; SAINFORT, F.; BARNARD, L.; EDWARDS, P. J.; EMERY, V. K.; KONGNAKORN, T.; MOLONEY, K. P.; ZORICH, B. S.: Older adults and visual impairment: what do exposure times and accuracy tell us about performance gains associated with multimodal feedback? In: *CHI '03: Proceedings of the SIGCHI conference on Human factors in computing systems*. New York, NY, USA: ACM Press, 2003, S. 33–40

Jain 2003
JAIN, Anil K.: Multimodal user interfaces: who's the user? In: *ICMI '03: Proceedings of the 5th international conference on Multimodal interfaces*. New York, NY, USA: ACM Press, 2003, S. 1–1

Kaiser et al. 2003
KAISER, E.; OLWAL, A.; MCGEE, D.; BENKO, H.; CORRADINI, A.; LI, X.; COHEN, P.; FEINER, S.: Mutual disambiguation of 3D multimodal interaction in augmented and virtual reality. In: *ICMI '03: Proceedings of the 5th international conference on Multimodal interfaces*. New York, NY, USA: ACM Press, 2003, S. 12–19

Kopp et al. 2004
KOPP, Stefan; TEPPER, Paul; CASSELL, Justine: Towards integrated microplanning of language and iconic gesture for multimodal output. In: *ICMI '04: Proceedings of the 6th international conference on Multimodal interfaces*. New York, NY, USA: ACM Press, 2004, S. 97–104

Kumar et al. 2004
KUMAR, Sanjeev; COHEN, Philip R.; COULSTON, Rachel: Multimodal interaction under exerted conditions in a natural field setting. In: *ICMI '04: Proceedings of the 6th international conference on Multimodal interfaces*. New York, NY, USA: ACM Press, 2004, S. 227–234

Lisetti & Nasoz 2002
LISETTI, Christine L.; NASOZ, Fatma: MAUI: a multimodal affective user interface. In: *MULTIMEDIA '02: Proceedings of the tenth ACM international conference on Multimedia*. New York, NY, USA: ACM Press, 2002, S. 161–170

Malaka et al. 2004
MALAKA, Rainer; HÄUSSLER, Jochen; ARAS, Hidir: SmartKom mobile: intelligent ubiquitous user interaction. In: *IUI '04: Proceedings of the 9th international conference on Intelligent user interface*. New York, NY, USA: ACM Press, 2004, S. 310–312

Marx & Schmandt 1994
MARX, Matt; SCHMANDT, Chris: Putting people first: specifying proper names in speech interfaces. In: *UIST '94: Proceedings of the 7th annual ACM symposium on User interface software and technology*. New York, NY, USA: ACM Press, 1994, S. 29–37

Mayhew 1999
MAYHEW, D. J.: *The Usability Engineering Lifecycle*. San Francisco: Morgan Kaufmann, 1999

McCaffery et al. 1998
MCCAFFERY, Fergal; MCTEAR, Michael F.; MURPHY, Maureen: A Multimedia Interface for Circuit Board Assembly. In: *Multimodal Human-Computer Communication, Systems, Techniques, and Experiments*. London, UK: Springer-Verlag, 1998, 213–230

Microsoft
Microsoft: *MiPad: Speech Powered Prototype to Simplify Communication Between Users and Handheld Devices*. `http://www.microsoft.com/presspass/features/2000/05-22mipad.asp`, retrieved on: 20.06.2008

Miyazaki 2002
MIYAZAKI, J.: *Discussion Board System with modality variation: From Multimodality to User Freedom*, Tampere University, Masterarbeit, 2002

Murray et al. 1993
MURRAY, A. C.; FRANKISH, C. R.; JONES, D. M.: Data-entry by voice: Facilitating correction of misrecognitions. In: BABER, C. (ed.); NOYES, J.M. (ed.): *Interactive Speech Technology: Human Factors issues in the Application of Speech Input/Output to Computers*. Bristol, PA: Taylor and Francis, 1993, S. 137–144

Nasoz et al. 2002
NASOZ, Fatma; OZYER, Onur; LISETTI, Christine L.; FINKELSTEIN, Neal: Multimodal affective driver interfaces for future cars. In: *MULTIMEDIA '02: Proceedings of the tenth ACM international conference on Multimedia*. New York, NY, USA: ACM Press, 2002, S. 319–322

Neely 1956
NEELY, K. K.: Effect of visual factors on the intelligibility of speech. In: *Journal of the Acoustical Society of America* 28 (1956), S. 1275–1277

Neuss 2001
NEUSS, Robert: *Usability Engineering als Ansatz zum Multimodalen Mensch-Maschine-Dialog*, Fakultät für Elektrotechnik und Informationstechnik, Technische Universität München, Diss., 2001

Niedermaier 2003
NIEDERMAIER, Franz B.: *Entwicklung und Bewertung eines Rapid-Prototyping Ansatzes zur multimodalen Mensch-Maschine-Interaktion im Kraftfahrzeug*, Fakultät für Elektrotechnik und Informationstechnik der Technischen Universität München, Diss., 2003

Obrenović et al. 2007
OBRENOVIĆ, Z.; ABASCAL, J.; STARČEVIĆ, D.: Universal accessibility as a multimodal design issue. In: *Commun. ACM* 50 (2007), Nr. 5, S. 83–88

Oviatt et al. 1999
OVIATT, S.; BERNARD, J.; LEVOW, G.: Linguistic adaptation during error resolution with spoken and multimodal systems. In: *Language and Speech (special issue on Prosody and Conversation)* 41 (1999), Nr. 3–4, S. 415–438

Oviatt et al. 2000
OVIATT, S.; COHEN, P.; WU, L.; VERGO, J.; DUNCAN, L.; SUHM, B.; BERS, J.; HOLZMAN, T.; WINOGRAD, T.; LANDAY, J.; LARSON, J.; FERRO, D.: Designing the User Interface for Multimodal Speech and Pen-based Gesture Applications: State-of-the-Art Systems and Future Research Directions. In: *Human Computer Interaction* 15 (2000), Nr. 4, S. 263–322

Oviatt & vanGent 1996
OVIATT, S.; VANGENT, R.: Error reslution during multimodal human-computer interaction. In: *Proc. of the International Conference on Spoken Language Processing* Bd. 2, 1996, S. 204–207

Oviatt 1999
OVIATT, Sharon L.: Mutual disambiguation of recognition errors in a multimodal architecture. In: *CHI '99: Proceedings of the SIGCHI conference on Human factors in computing systems*. New York, NY, USA: ACM, 1999, S. 576–583

Oviatt 2000a
OVIATT, Sharon L.: Multimodal Signal Processing in Naturalistic Noisy Environments. In: YUAN, B. (ed.); HUANG, T. (ed.); TANG, X. (ed.): *Proceedings of the 6th International Conference on SPoken Language Processing (ICSLP)* Bd. 2. Peking: Chinese Friendship Publishers, 2000, 696–699

Oviatt 2000b
OVIATT, Sharon L.: Multimodal system processing in mobile environments. In: *UIST '00: Proceedings of the 13th annual ACM symposium on User interface software and technology*. New York, NY, USA: ACM Press, 2000, S. 21–30

Oviatt 2000c
OVIATT, Sharon L.: Taming recognition errors with a multimodal interface. In: *Commun. ACM* 43 (2000), Nr. 9, S. 45–51

Oviatt & Kuhn 1998
OVIATT, Sharon L.; KUHN, Karen: Referential features and linguistic indirection in multimodal language. In: *Proceedings of the International Conference on Spoken Language Processing* Bd. 6, ASSTA, 1998, 2339–2342

Patterson & Gowdy 2003
PATTERSON, E.K.; GOWDY, J.N.: An audio-visual approach to simultaneous-speaker speech recognition. In: *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings* Bd. 5, 2003, 780–783

Pieraccini et al. 2004
PIERACCINI, R.; DAYANIDHI, K.; BLOOM, J.; DAHAN, J.-G.; PHILLIPS, M.; GOODMAN, B. R.; PRASAD, K. V.: Multimodal conversational systems for automobiles. In: *Commun. ACM* 47 (2004), Nr. 1, S. 47–49

Ratzka 2008
RATZKA, Andreas: Design Patterns in the Context of Multi-modal Interaction. In: *To appear in: Proceedings of the 6th Nordic Conference on Pattern Languages of Programs 2007 VikingPLoP 2007*, 2008

Ratzka & Wolff 2006
RATZKA, Andreas; WOLFF, Christian: A Pattern-based Methodology for Multimodal Interaction Design. In: SOJKA, P. (ed.); KOPEČEK, I. (ed.); PALA, K. (ed.): *Proc. of Text, Speech, and Dialogue, TSD'06*. Berlin, Heidelberg: Springer, 2006 (LNAI 4188), S. 677–686

Reithinger et al. 2003
REITHINGER, N.; ALEXANDERSSON, J.; BECKER, T.; BLOCHER, A.; ENGEL, R.; LÖCKELT, M.; MÜLLER, J.; PFLEGER, N.; POLLER, P.; STREIT, M.; TSCHERNOMAS, V.: SmartKom: adaptive and flexible multimodal access to multiple applications. In: *ICMI '03: Proceedings of the 5th international conference on Multimodal interfaces*. New York, NY, USA: ACM Press, 2003, S. 101–108

Saenko et al. 2004
SAENKO, Kate; DARRELL, Trevor; GLASS, James R.: Articulatory features for robust visual speech recognition. In: *ICMI '04: Proceedings of the 6th international conference on Multimodal interfaces*. New York, NY, USA: ACM Press, 2004, S. 152–158

Schomaker et al. 1995
SCHOMAKER, L.; NIJTMANS, J.; CAMURRI, A.; LAVAGETTO, F.; MORASSO, P.; BENOÎT, C.; GUIARD-MARIGNY, T.; GOFF, B. L.; ROBERT-RIBES, J.; ADJOUDANI, A.; DEFÉE, I.; MÜNCH, S.; HARTUNG, K.; BLAUERT, J.: A Taxonomy of Multimodal Interaction in the Human Information Processing System. Version: Februar 1995. `http://vonkje.cogsci.kun.nl/ miami/reports/reports.html`. 1995. (A Report of the Esprit Basic Research Action 8579 MIAMI). – Forschungsbericht

Sinnig et al. 2004
SINNIG, Daniel; GAFFAR, Ashraf; REICHART, Daniel; SEFFAH, Ahmed; FORBRIG, Peter: Patterns in Model-Based Engineering. In: *CADUI*, 2004, S. 195–208

Snelick et al. 2003
SNELICK, Robert; INDOVINA, Mike; YEN, James; MINK, Alan: Multimodal biometrics: issues in design and testing. In: *ICMI '03: Proceedings of the 5th international conference on Multimodal interfaces*. New York, NY, USA: ACM Press, 2003, S. 68–72

Suhm et al. 2001
SUHM, B.; MYERS, B.; WAIBEL, A.: Multimodal error correction for speech user interfaces. In: *ACM Trans. Comput.-Hum. Interact.* 8 (2001), Nr. 1, S. 60–98

Sumby & Pollack 1954
SUMBY, W. H.; POLLACK, I.: Visual contribution to speech intelligibility in noise. In: *Journal of the Acoustical Society of America* 26 (1954), S. 212–215

Summerfield 1979
SUMMERFIELD, A. Q.: Use of visual information for phonetic perception. In: *Phonetica* 36 (1979), S. 314–331

Tan et al. 2003
TAN, Yeow K.; SHERKAT, Nasser; ALLEN, Tony: Error recovery in a blended style eye gaze and speech interface. In: *ICMI '03: Proceedings of the 5th international conference on Multimodal interfaces*. New York, NY, USA: ACM Press, 2003, S. 196–202

Tanaka 1999
TANAKA, Katsumi: A robust selection system using real-time multi-modal user-agent interactions. In: *IUI '99: Proceedings of the 4th international conference on Intelligent user interfaces*. New York, NY, USA: ACM Press, 1999, S. 105–108

Tidwell 1999
TIDWELL, J.: *COMMON GROUND: A Pattern Language for Human-Computer Interface Design*. `http://www.mit.edu/~jtidwell/common_ground.html`. Version: 1999, retrieved on: 20.06.2008

Tidwell 2005
TIDWELL, Jenifer: *Designing Interfaces: Patterns for Effective Interaction Design*. O'Reilly, 2005

Trabelsi et al. 2002
TRABELSI, Z.; CHA, S.-H.; DESAI, D.; TAPPERT, C.: A voice and ink XML multimodal architecture for mobile e-commerce systems. In: *WMC '02: Proceedings of the 2nd international workshop on Mobile commerce*. New York, NY, USA: ACM Press, 2002, S. 100–104

Vitense et al. 2002
VITENSE, H. S.; JACKO, J. A.; EMERY, V. K.: Multimodal feedback: establishing a performance baseline for improved access by individuals with visual impairments. In: *Assets '02: Proceedings of the fifth international ACM conference on Assistive technologies*. New York, NY, USA: ACM Press, 2002, S. 49–56

Wachsmuth 2001
WACHSMUTH, Sven: *Multi-modal Scene Understanding Using Probabilistic Models*, Technischen Fakultät, Universität Bielefeld, Diss., 2001

Wahlster et al. 2001
WAHLSTER, W.; REITHINGER, N.; BLOCHER, A.: SmartKom: Towards Multimodal Dialogues with Anthropomorphic Interface Agents. In: WOLF, G. (ed.); KLEIN, G. (ed.); Projektträger des BMBF für Informationstechnik: Deutsches Zentrum für Luft- und Raumfahrttechnik (DLR) e.V. (Veranst.): *Proceedings of International Status Conference: Lead Projects Human-Computer-Interaction*. Saarbrücken, 2001, 23–32

Wasinger 2006
WASINGER, Rainer: *Multimodal Interaction with Mobile Devices: Fusing a Broad Spectrum of Modality Combinations*. Saarbrücken, Naturwissenschaftlich-Technische Fakultät I der Universität des Saarlandes, Diss., 2006

van Welie & Trætteberg 2000
WELIE, M. van; TRÆTTEBERG, H.: Interaction Patterns in User Interfaces. In: *Proceedings of the Seventh Pattern Languages of Programs Conference*. Monticello, Illinois, USA, 2000

Yang et al. 1998
YANG, J.; STIEFELHAGEN, R.; MEIER, U.; WAIBEL, A.: Visual tracking for multimodal human computer interaction. In: *CHI '98: Proceedings of the SIGCHI conference on Human factors in computing systems*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1998, S. 140–147

Yang et al. 1999
YANG, J.; ZHU, X.; GROSS, R.; KOMINEK, J.; PAN, Y.; WAIBEL, A.: Multimodal people ID for a multimedia meeting browser. In: *MULTIMEDIA '99: Proceedings of the seventh ACM international conference on Multimedia (Part 1)*. New York, NY, USA: ACM Press, 1999, S. 159–168

Zeng et al. 2004
ZENG, Z.; TU, J.; LIU, M.; ZHANG, T.; RIZZOLO, N.; ZHANG, Z.; HUANG, T. S.; ROTH, D.; LEVINSON, S.: Bimodal HCI-related affect recognition. In: *ICMI '04: Proceedings of the 6th international conference on Multimodal interfaces*. New York, NY, USA: ACM Press, 2004, S. 137–143

Zhang et al. 2004
ZHANG, Q.; IMAMIYA, A.; GO, K.; MAO, X.: Overriding errors in a speech and gaze multimodal architecture. In: *IUI '04: Proceedings of the 9th international conference on Intelligent user interface*. New York, NY, USA: ACM Press, 2004, S. 346–348

Turning me on, turning me off

More patterns for a pattern language
of interactive information graphics

Christian Kohls, c.kohls@iwm-kmrc.de
Knowledge Media Resarch Centre, Tuebingen

Tobias Windbrake, wb@fh-wedel.de
University of Applied Sciences Wedel, Wedel

Abstract

Interactive graphics are an effective way of communication and information delivery, especially for complex domains. However, domain experts are rarely aware of the potentials of interactive visual displays and which interaction principles can be in charge for communication and teaching purposes. In this paper we extend a pattern language for interactive information graphics and present four patterns. These patterns are all variations of buttons that can switch between two visual states. Each pattern describes the consequences and special fields of application related to the chosen button type in an educational setting.

Target Group

The language captures expertise about interactive graphics and focuses on the educational benefits gained by each form of interaction. The primary target group are developers of multimedia learning materials – for both, classroom and online sessions. The language can be particularly useful for:

Educators who want to enhance their course materials with interactive graphics.
Teachers who want to use interactive content in their classroom lessons.
Editors and designers of rich online learning materials.
Information designers who want to enrich websites with interactive diagrams.
Exhibitors who want to design interactive information kiosks in museums or visitor centres.
Users of interactive displays.
Programmers who want to learn when and why an interactive screen adds value to content presentation.
Project teams who need a common language to plan their development of materials, e.g. course designers can unambiguously communicate to developers.

The goal is to illustrate what forms of interactive graphics are used in today's multimedia materials and how and when they can be used beneficial in educational contexts.

About the patterns

Interactive graphics certainly helps to analyze, understand and communicate models, concepts and data [Rie90], [SM00], [May01] if applied appropriately. Each pattern explicitly names situations where to use it and gives a rationale why it can support the communication process. With respect to the aimed target group of this pattern collection, the technological aspects of the pattern descriptions should be low-threshold. There is no implementation section in the documented patterns because educators, teachers and trainers are not likely to implement the patterns in program code. Skilled programmers on the other hand will profit more from understanding the pedagogical values than seeing rather trivial example codes. Another issue is that implementations in different authoring tools are very idiomatic. Thus, explaining the implementation in one tool would give no insights into the options other authoring tools may offer. Fortunately, multimedia authoring tools recently started to offer templates, wizards or objects to instantiate the patterns without programming. The major goal of this language is to inform about the potentials, benefits and limitations of interactive graphics. The proper use of interactive graphics is another issue. Using an inappropriate form of interaction can do harm to the process of learning and understanding. Therefore, this pattern collection is also a guide to choose the right formats for the tasks at hand.

The patterns presented in this paper are different variations of graphic buttons. Buttons are common widgets in user interface design. The perspective taken in the present patterns is to discuss the educational use of buttons in interactive graphics. One can also learn about the consequences and special fields of application depending on the special type of button that is chosen. One of the major differences to other descriptions of buttons is that in the given examples the visual form of the buttons is different from the standard box layouts. In educational illustrations any graphical form, shape, text, area or image can be used as a button.

The pattern collection for interactive information graphics relates or overlaps with languages for human computer interaction [Ba97], [Bor01], [GLC01], [MJ98], [Tid05], [Wel05], the design of websites [WV03], [Sc05], [Mah06], [MLC05], [DLH04] and patterns for education and e-learning [PPP], [VW04], [ND05]. The patterns that can be found in this paper are all well-known human-computer-interaction patterns (and are described in [Tid05], [Wel05]). What is new to the patterns is the view of how to use these interaction forms in an educational contexts and which pedagogical challenges are solved by the patterns. The patterns presented are a prosecution of a pattern collection that has been presented on previous PLoPs. As a feedback from the various workshops, the pattern descriptions have become highly visual.

PLoP 2006	EuroPLoP 2007	PLoP 2007	EuroPLoP 2008
OPTIMIZE OBJECT PERCEPTION	TRANSPORT OBJECTS	ACTIVE AREAS (HOT AREAS)	ON/OFF BUTTON
DYNAMIC LABELLING	SYNCHRONIZE OBJECT MOTION	ACTIVATOR	ROLLOVER BUTTON
SWITCH BETWEEN OBJECT STATES	DRAG RESTRICTION	NO-GO-AREAS	RADIO BUTTON
		SANDBOX	INFORMATION DISPLAY

Note that the patterns TOGGLE BUTTON, ROLLOVER BUTTON, RADIO BUTTON, and INFORMATION DISPLAY are differentiations of the SWITCH BETWEEN OBJECT STATES pattern presented at PLoP 2006 [KW06]. The SWITCH BETWEEN OBJECT STATES focussed on technical similarities. On the code level the four patterns presented in this paper are almost identical, i.e. in Java the difference between a TOGGLE and a ROLLOVER BUTTON is usually just in which method of the event interface the property is changed. However, end users perceive mouse over and mouse clicks quite differently in terms of usability and semantics. In an experimental setting we have asked teacher students to categorize interactive graphics based on some patterns of this pattern language [KU09]. For all students the event trigger (mouse over or mouse click) was a discriminating feature.

Also note that DYNAMIC LABELLING is a pattern of a higher level focussing on educational function only rather than the interaction form. A DYNAMIC LABEL can be implemented by all of the patterns present in this paper and by ACTIVE AREAS and ACTIVATORS as well.

Different buttons for different purposes

Now, which button type is the best for you? It depends on the context and type of information you intend to provide. What you can do with each type of button you will find in the "What can I do with this interaction form?" sections which list typical scenarios and applications. Every button type has its own benefits and liabilities. To select multiple elements simultaneously one can use TOGGLE BUTTONS E or several independent groups of RADIO BUTTONS. An TOGGLE BUTTON lets the user decide at which time to switch each element individually. While this adds more freedom to the user, it can also result in messed up screens, e.g. too many added elements increases complexity, pop-ups hide other information or the whole graphic alters to an unsatisfactory state. To ensure that graphic elements automatically switch back to their original states, ROLLOVER and RADIO BUTTONS are a good choice. ROLLOVERS are more intuitive, give faster access to extra information and implicitly indicate which element is currently inspected while RADIO BUTTONS keep selected information permanently available and do not block the mouse pointer for other tasks. TOGGLE BUTTONS can build up graphics step-by-step whereas the other interaction forms allow transferring a focus step-by-step.

TOGGLE, ROLLOVER, and RADIO BUTTONS all alter a base graphic. That way, newly added information is integrated into the graphic but also hides or overwrites information. In opposition, an INFORMATION DISPLAY is always spatially separated and does not interfere with the graphic. But it also splits the attention between inspected elements and the explanatory information. Only INFORMATION DISPLAYS allow defining more than two visual states for an element. The element that shows the state for the INFORMATION DISPLAY is always distant to the display itself. All other button types allow to be both trigger and output element in one. Which functions are supported by which button type is summarized in the following table:

	Toggle	Rollover	Radio	Info Display (click/rollover)
Selection of multiple elements simultaneously (e.g. to compare or filter objects)	X			
Assigning multiple (>2) visual states to an element				X
Mouse pointer indicates which element is currently inspected		X		- / X
Mouse pointer is available for other tasks (and not stressed for activation)	X		X	X / -
Build up a graphic step by step	X			
Focus on graphic elements step by step		X	X	X
Additional elements do not overlap base information permanently		X	X	X
Additional elements are nearest to base information	X	X	X	
Allows to change components of a graphic	X	X	X	
Added elements and element changes remain persistent until explicitly reset	X		X	X / -
Elements do not need a visual affordance to indicate that they can be clicked		X		- / X
Highlight and focus elements	X	X	X	
Button element (hot area) can be changed itself	X	X	X	
Can be used to replace large components or the complete graphic	X			X
Can be used to replace small components of a graphic	X	X	X	

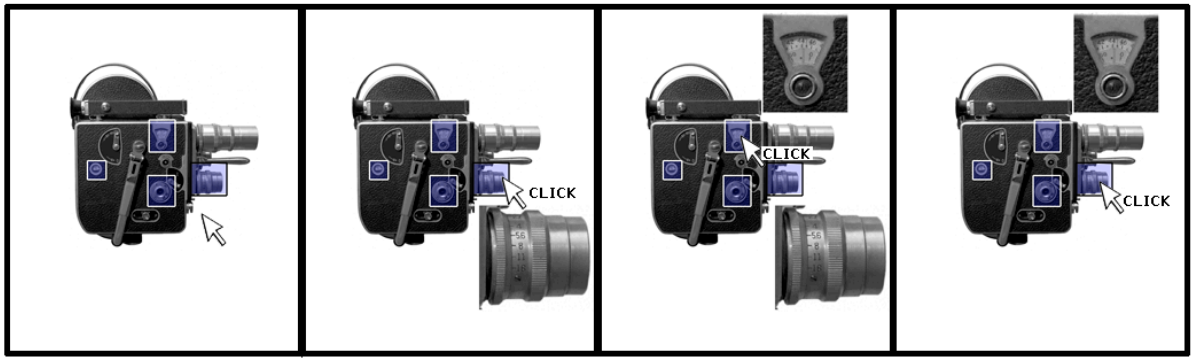
Supports exploration of a graphic	X	X		X
Supports explicit selection of elements	X		X	
No explicit operation required to deactivate an element		X	X	- / X

Toggle Button – ON/OFF Button

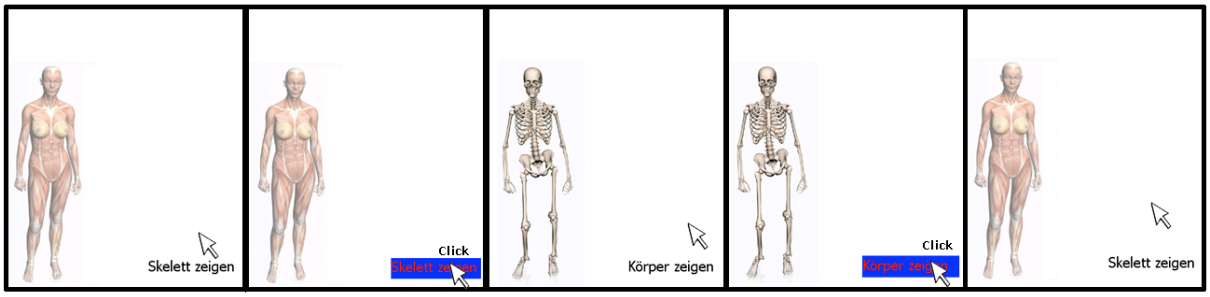
Provide a way to change between two visual states of an element by clicking at it or a separate button.

Examples

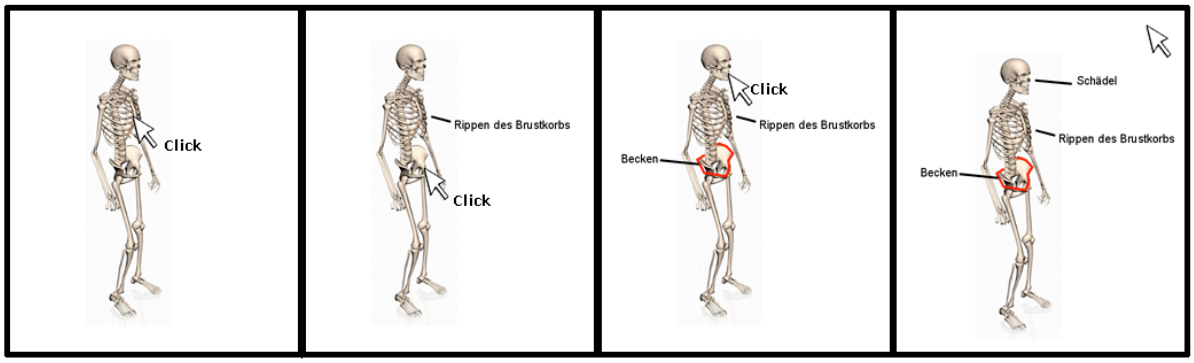
Zoom on Demand: If the user clicks at on of the marked areas, an enlarged illustration pops out. The detailed illustration has to be deactivated explicitly by clicking at the corresponding area for another time.



Switch illustration: The text acts as a button that switches between a medical illustration of muscles or skeleton.



Dynamic labels: By clicking at areas of the skeleton, the labels appear.



Context

For illustrative purposes you are using an information graphic in your classroom or in multimedia materials. The information graphic is very complex and consists of many visual elements (e.g. labels, boxes, parts, connections). Some of the elements might be of a dynamic nature and can change their states (e.g. a light bulb can be on off, water can be frozen or not, an engine can idle or run). Some elements are in themselves very complex and need further explanation or even different kinds of explanations. Adding all these multiple states and multiple explanations makes the graphic even more complex. Yet all the information given is needed and cannot be reduced without loss.

Problem

You cannot leave out elements without losing important information necessary to understand the illustrated subject but at the same time the complexity and amount of information is just too overwhelming and makes it very hard for students to process the given information and focus on relevant aspects.

Forces

Many subjects require complex graphics to illustrate relations and contextual information but huge graphics can be overwhelming and the observer may not know where to start.

Today's presentation tools allow adding information in a predefined sequence step-by-step, however, depending on the subject it may be desirable to add elements in random order.

Adding information allows integrating new elements into the existing knowledge representation but what to do with information that is needed only temporarily?

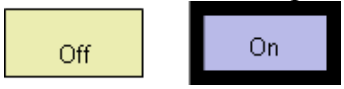
Multiple representations of objects (i.e. to show transformations, causal chains, different viewpoints or levels of detail) can be accessed by navigating to another graphic screen but what if only parts of the graphic should change? How to keep the number of prepared graphics small if different aspects of a graphic can vary independently?

Solution

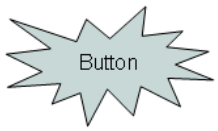
Add, remove or change elements dynamically in one graphic to reduce its complexity and show elements or their alternative states only when needed.

To exactly control when elements change their state from invisible to visible or from inactive to activate or from on to off, define for each alterable element a trigger. The trigger changes the alterable element between two preset states. We shall call these two states ON and OFF.

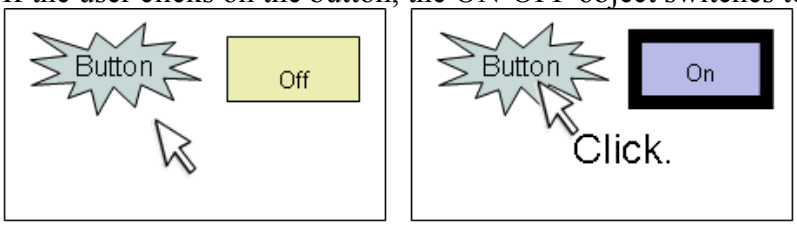
Each of the states is represented by different visual properties:



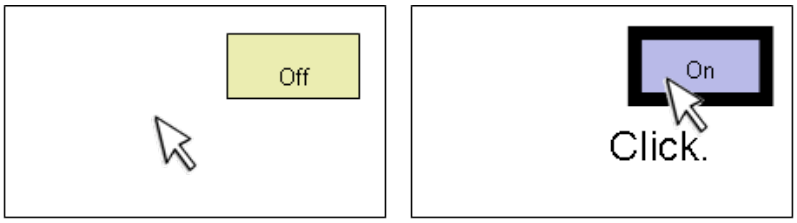
Define an element on the screen as a button that triggers the state change:



If the user clicks on the button, the ON-OFF object switches to the opposite state:

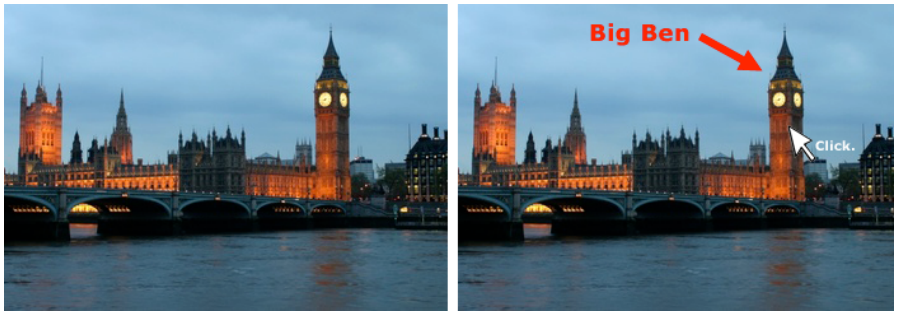


The button can be the ON-OFF element itself. That is, a click on the elemrnz will change its own visual state:

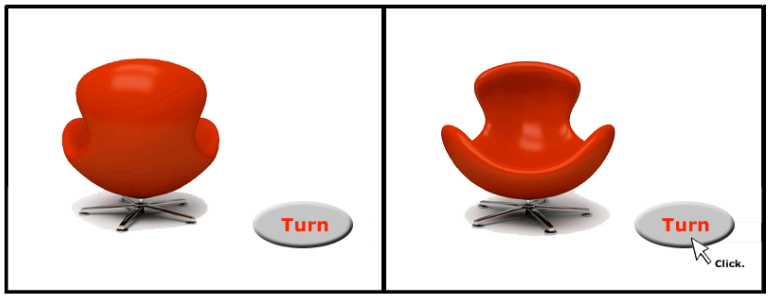


Details

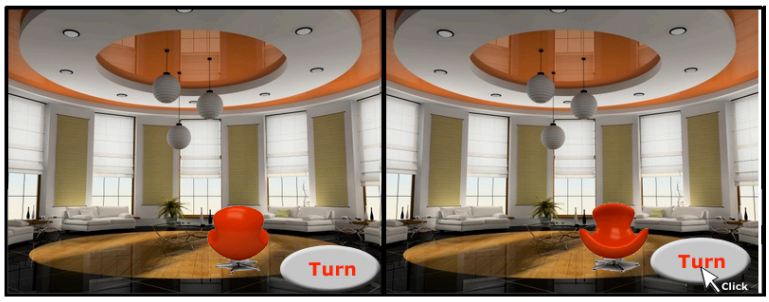
Elements that change their state between visible and invisible overlay an image in the background if they become visible. In this case extra information is added to a base graphic. In the following example, a transparent button element overlays Big Ben. If the user clicks at it, arrow and annotation overlay the picture in the background without changing the picture:



Instead of adding elements, buttons can be used to change elements. In the next example the "Turn"-button changes the image element of the chair:



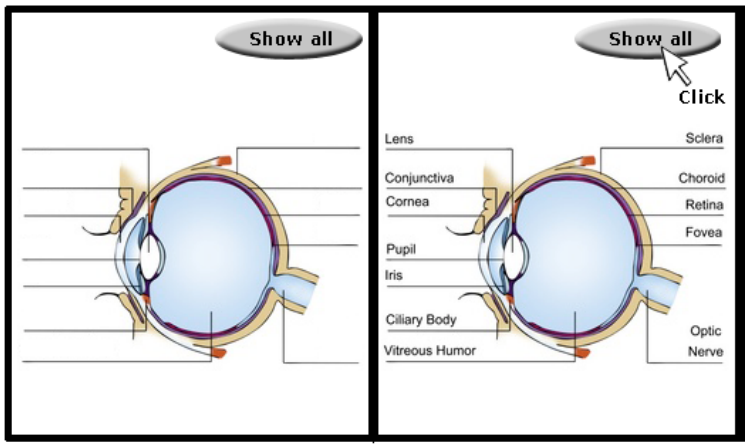
Buttons can be integrated in a larger picture or illustration. By splitting up an image into independent elements, each element can switch between ON/OFF states if required. The next example shows the chair as a changeable element of a picture:



Instead of clicking an extra “Turn” button, the element could be buttons itself:

The great advantage of ON/OFF buttons is that the user has full control over the activation of elements. The order of adding or changing elements is not predefined. Also, the user is free to choose how long an element remains in a chosen state. He can activate any selection of ON/OFF elements whenever he wants and set the complexity of the shown graphic as he wish. Information can be filtered and the combination of activated elements is not preset.

An ON/OFF button can turn multiple elements to ON or OFF in one step:



Other properties that typically change are colour or text. Colour changes can be used to highlight some elements out of a group. Text changes can be used to give different information, e.g. translation of a vocabulary:

What can I do with this interaction form?

- Switch on and off additional information on the screen.
- Show pop-up information in small boxes on demand.
- Show the front and the back of an object.
- Switch on and off overlay elements for pictures.
- Switch on and off labels that explain parts of a picture.
- Ask a question and show the answer by clicking at the object.
- Reduce complexity by hiding elements if not needed.
- Adding elements step by step in random order.

Rationale

Adding visual elements on demand reduces the complexity as only the elements currently needed are displayed. Also, it allows building up complex graphics step-by-step and helps to avoid cognitive overload by chunking information [CS91]. Having the user decide which elements should be turned on or off at any time gives maximum control to the user. In general, buttons are graphic user interface elements that can trigger action and display states as well [Tid05]. In the context of educational graphics they switch between two states of elements. Thus, all objects of the world that may be represented by two major states, can be simulated by an ON/OFF button. Switching the states explicitly by a mouse click is a form of direct manipulation [Shn87].

Drawbacks

Each button can only set two states, ON and OFF. Many objects in the real world, however, offer more than two interesting visual states. To assign multiple visuals states to an object, an INFORMATION DISPLAY can be used.

The user has to explicitly turn off an element to deactivate it. In particular, if too many elements are activated simultaneously, an image soon becomes overloaded. RADIO BUTTONS and ROLLOVER buttons automatically turn OFF an activated element for the price of having only one element activated at a time.

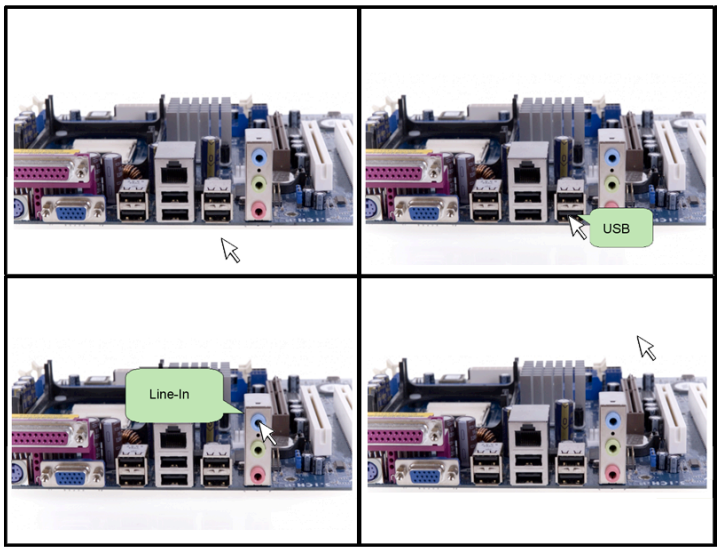
The changeable graphic elements have to be equipped with their own affordances in such a way that the user can understand where he can click.

Rollover Button

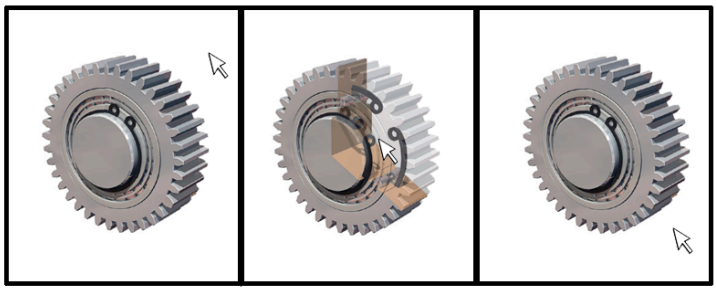
Provide a way to switch between two visual states of an object when the mouse pointer enters a hot area.

Examples

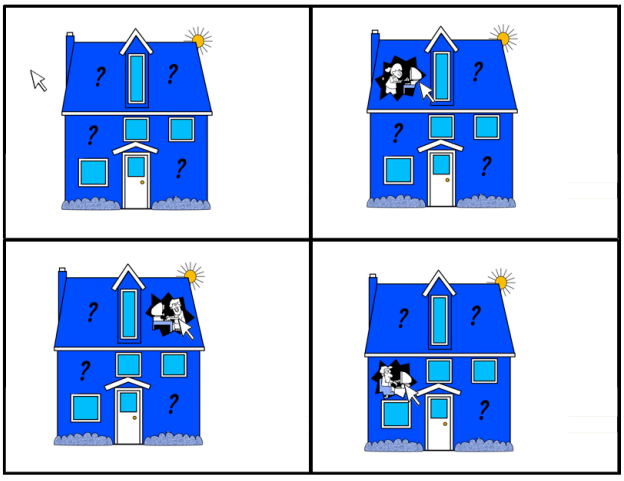
Tips/Hints: If the mouse pointer moves over interesting areas of the image, a text bubble appears. The text in the bubble explains what is shown under the mouse pointer. The text bubbles disappear automatically as the mouse moves on to other interesting areas.



Ghost drawing on demand: If the mouse pointer moves over certain parts of the image, an overlay shows the insights of the object.



Look through walls: If the mouse pointer moves over the question marks, the wall disappears and the user gains insights into the house.



Context

You are using an information graphic that can be explored by your students individually on a laptop or PC. The information graphic consists of many elements and some elements are interrelated (e.g. labels and text are related to objects, elements can be part of the same set, elements can be connected, elements can show different states of the same object). The interrelation between the elements should stand out and it should be clear which elements belong together. The student should be allowed to focus on some elements without losing the context of the whole graphic.

Problem

A huge number of elements makes it hard to recognize element relations and overloads the screen. Elements may interfere each other, hide other elements, or be hard to perceive as separate units.

Forces

Changing a graphic allows exploring given information but how can the user recognize which data s/he is currently focussing on?

To replace graphic elements or to open information pop-ups allows adding details on demand and fading-in further explanations but how can the user efficiently find out which parts of the graphic trigger such changes?

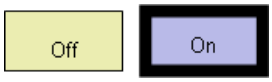
Adapting the given information according to the needs of the user allows for self-paced learning but altering all parts of the graphic can also mess up the illustration. Also, adding more and more information reduces the clearness of the original graphic.

Adding information ad-hoc to provide object specific information avoids split attention effects but the overlay information also hides some parts of the graphic in the background.

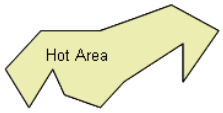
Solution

Add, remove or change elements dynamically by using the mouse pointer as an implicit trigger for element changes. This allows for rapid, user-controlled changes of focussed elements. If the mouse pointer points to a specific element, interrelated elements can change their states (e.g. fade in, highlight or show a different image). Automatically reset all changes if the mouse pointer exits the trigger element to no longer focus the activated element(s).

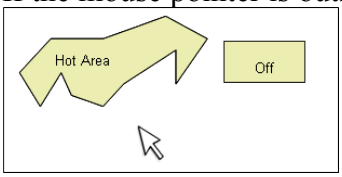
To highlight elements, define for the elements in question two visual states ON and OFF by having some visual properties set differently:



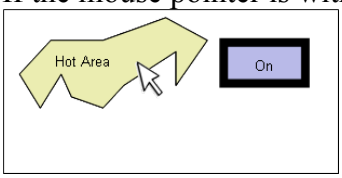
Define an element as a hot area that is sensitive to the mouse pointer:



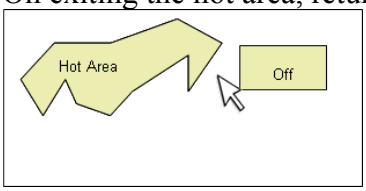
If the mouse pointer is outside the hot area, show the OFF state:



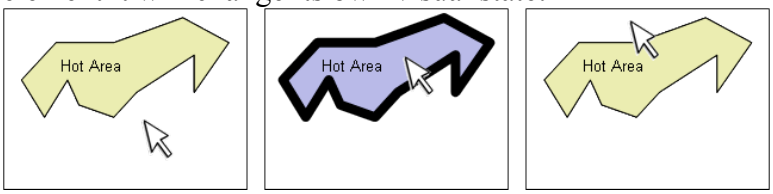
If the mouse pointer is within the hot area, show the ON state:



On exiting the hot area, return to the OFF state:



The hot area can be the ON-OFF element itself. That is, if the mouse pointer enters the element it will change its own visual state:



Details

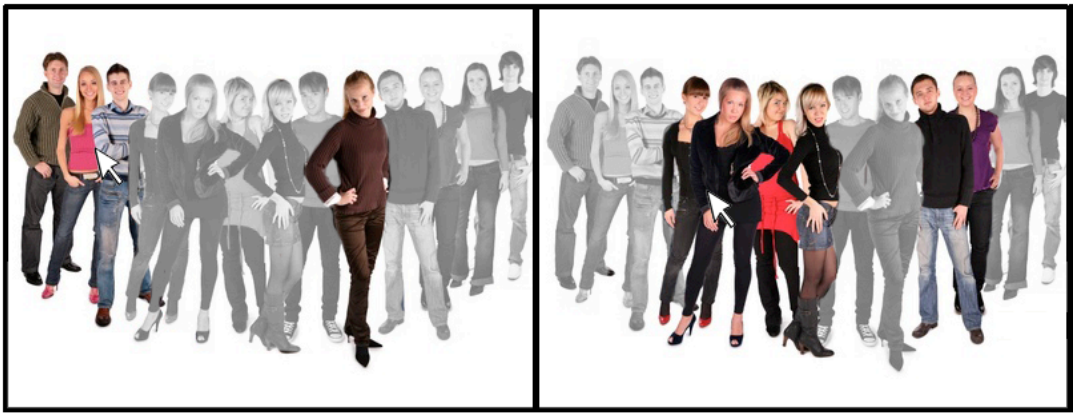
A rollover is a button that automatically turns an elemrnz ON if the mouse pointer enters the hot area element and automatically turns the same object OFF if the mouse pointer exits the hot area element.

Information changes or pops out implicitly without having the user to explicitly click at an element. The later case would require him to know where he can click, while a rollover offers the information more incidentally. Users can explore images intuitively and will immediately recognize where additional information is available. Information is presented rather on-focus than on-demand. If the mouse exits the hot area, the ON/OFF element returns to its visual OFF state. This means that there can be only one element highlighted at a time. This cleans up the screen automatically but also means that information is only temporarily available.

Because the ON state is only temporarily for an element, a roll over can indicate that an element or an area is activated at the moment. It can highlight which element is focussed by the mouse pointer. If elements on the screen offer an affordance for further operations (i.e. clicking at an element or drag the element), the highlighting indicates on which element the mouse pointer would operate.

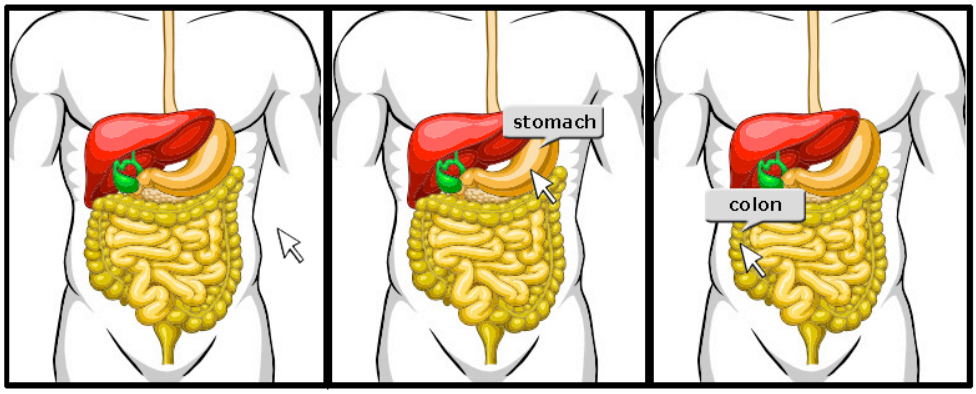
The transient character of rollover buttons can be useful to highlight related objects on the screen. For example, by moving the mouse pointer over one element of a set, all members of

the set could be highlighted. In the next example you see a group photo in which the members of a team are highlighted simultaneously:

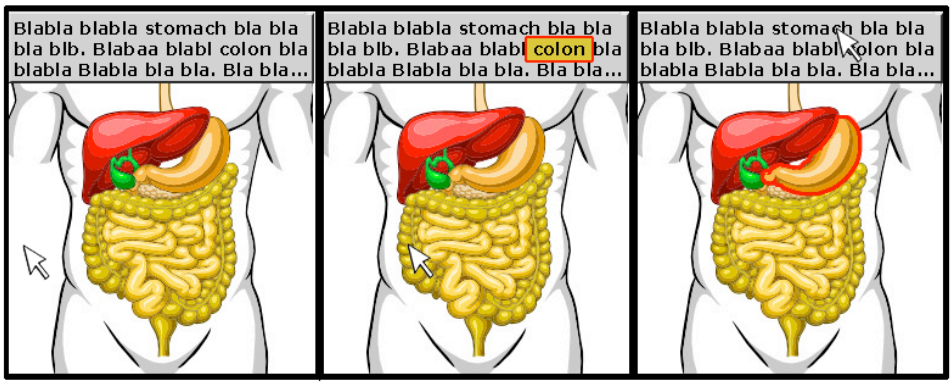


An advantage of an overlay (such as a label or text bubble) is the spatial nearness to the object. Hence, the coherence of base data and faded-in data is optimized. However, overlays are also disturbing because overlays hide data in the background.

A rollover minimizes the area which is hidden because the overlay disappears as soon as the mouse exits the hot area:



To avoid overlapping completely, an INFORMATION DISPLAY can be used for the price of separating text and image. To conserve coherence between an illustration and the text, corresponding words and objects could be highlighted at the same time:



What can I do with this interaction form?

- Highlight related areas or related information on the screen.
- Show or highlight all objects of one class or one set.
- Indicate that an object is activated or that the mouse points at an object that can be activated.
- Highlight the bounds of an object, e.g. highlight where the border of a country runs.
- Dynamically fade in info boxes, windows or text bubbles.
- Provide tool tips.
- Show objects in a different state by activating them by the mouse pointer.

Rationale

Roll-over buttons add and remove additional information automatically to the graphic. Thus, the visual elements are not required to have additional visual affordances [Gib79], because the mouse pointer can explore the graphic by moving over interesting areas. Because no explicit operations (such as mouse clicks) are required, the use of roll-over is self-explaining to the user. Adding information only temporarily will automatically clean up the graphic when the focus is lost. Therefore, information overlays can be integrated into the graphic with only minor disturbance. The advantage is that extra information is close to the visual context it

refers to. Hence, the contiguity principle - graphics and corresponding texts should be placed close together – is respected [MM99]. Also, having related objects close together lets the observer perceive them as a unit according to the gestalt law of proximity [Gol89]. By only having one roll-over button activated at a time, an implicit focus is given to direct the attention of the user. As attention is a limited resource, this can help to reduce the cognitive load [CM02]. Confusion about which label belongs to which area is avoided since the mouse pointer establishes an unambiguous point of reference.

Drawbacks

Using the mouse pointer for the activation of elements means that the pointer cannot be used for anything else while activating the specific information. To let information pop out for a longer time period, a RADIO BUTTON is an alternative which also takes care of only having one information unit activated at a time. A RADIO BUTTON, however, is less intuitive as it requires a more explicit user input (mouse clicks).

Some interactive displays, such as touch monitors and some interactive whiteboards, do not recognise mouse motion without pressing the surface. Hence a rollover (which occurs on moving the mouse without pressing the mouse button) will not work. As an option, one can design an ON/OFF button that responds to pressing the mouse button (instead of mouse entering) and releasing the mouse button (instead of mouse exiting).

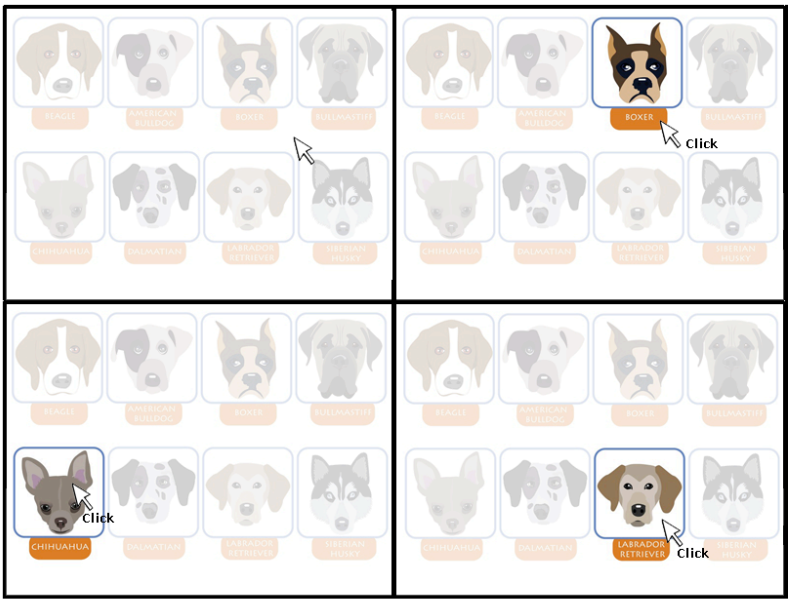
It is not possible to select multiple elements simultaneously (e.g. to compare representations or reduce the number of objects). Since each added information is automatically discarded if the focus is lost, a ROLLOVER can not be used to create complex illustrations step-by-step. To implement such behaviour, use a TOGGLE BUTTON instead.

Radio Button

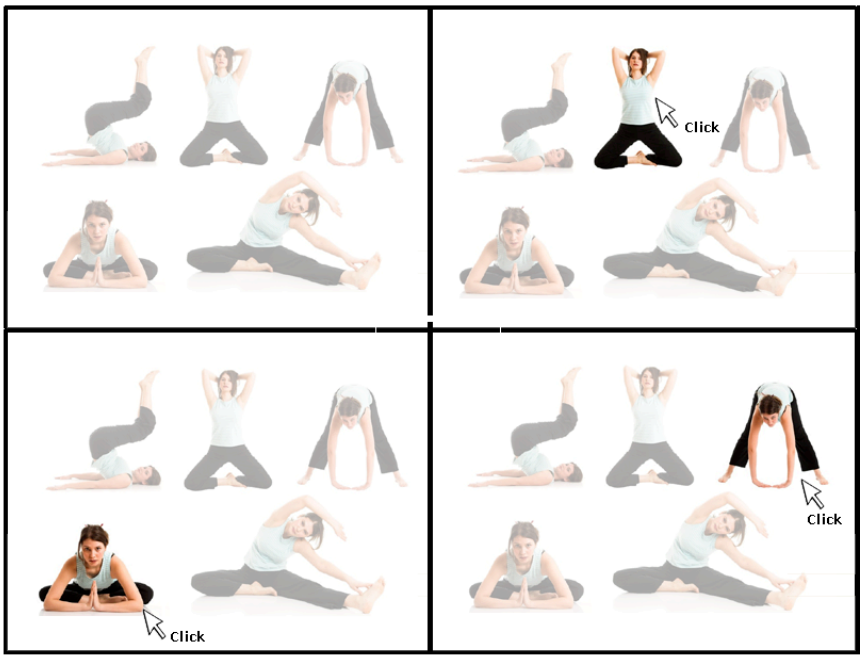
Provide a way to highlight one element out of a group.

Examples

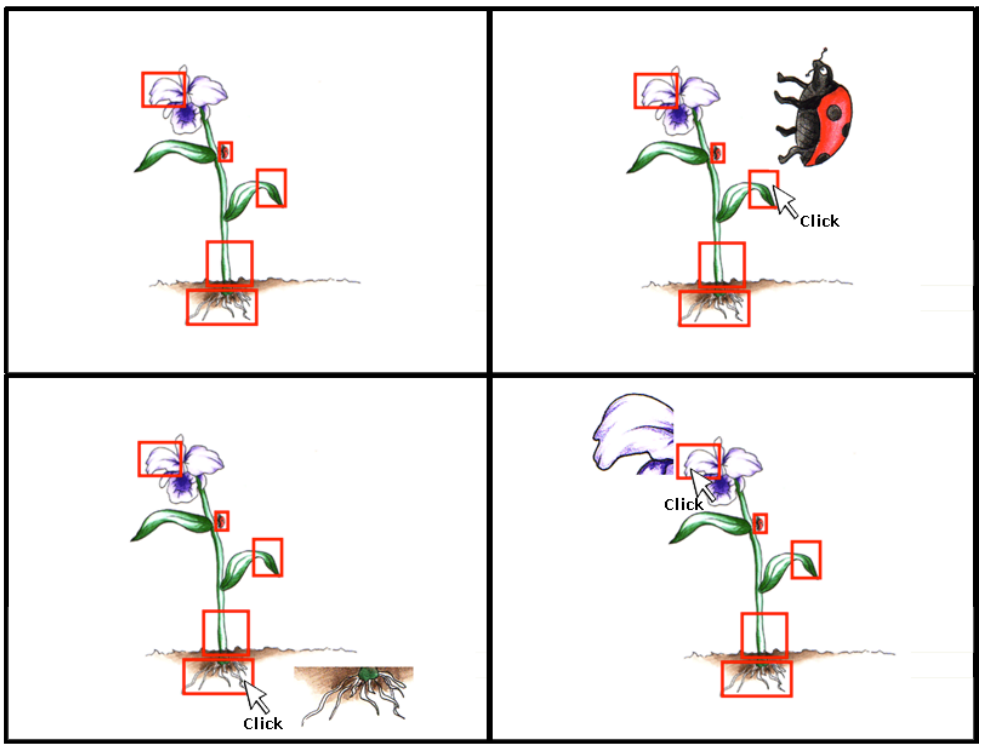
Focus on information. By clicking at one of the dogs, its opacity changes and the user can focus on that information unit. Only one of the information units is highlighted at a time.



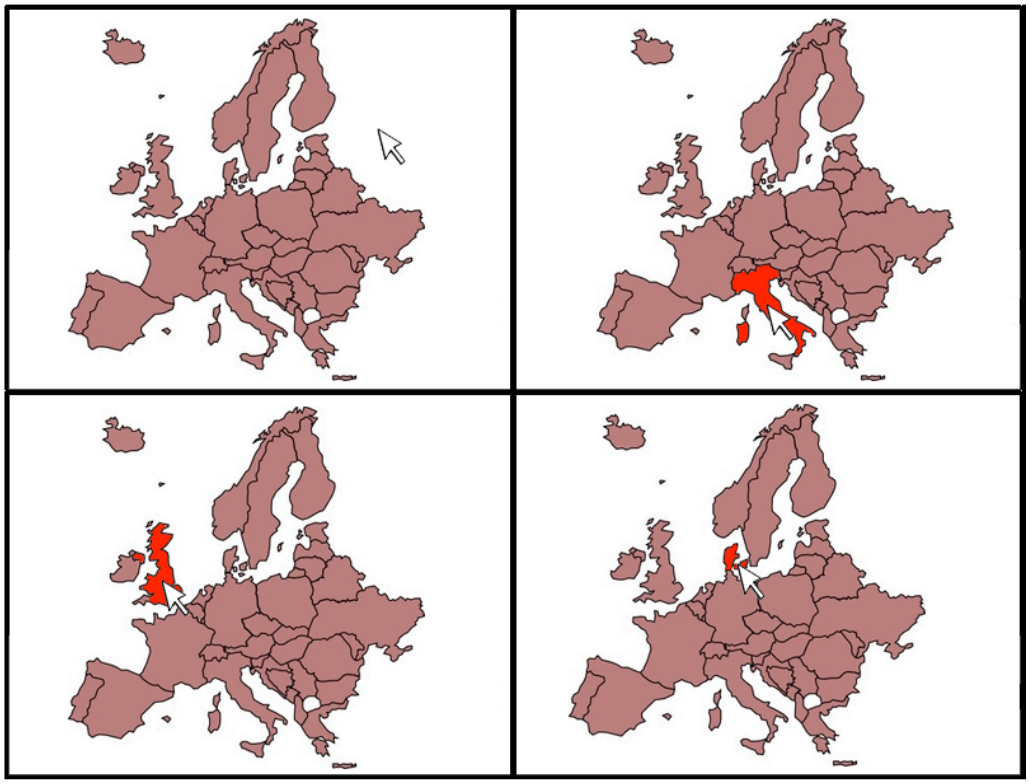
Yoga exercises: One of the exercises is highlighted to show the participants which exercise should be performed next.



Zoom on Demand: If the user clicks at on of the marked areas, an enlarged illustration pops out. Any enlarged illustration shown before disappears. Thus, only one zoom is visible at a time and the attention is directed to only one zoomed detail.



Showing bounds: If the user clicks at any country, the complete area of that country is highlighted. Thus, one can see exactly which parts belong to a country. In particular, this is helpful if the areas are separated.



Context

You are showing a complexly information graphic with distinct elements to your students. In your presentation you are going to focus on some of the elements separately. To assist your audience in paying attention to the element you are talking about it should be easy to find and recognize it.

Problem

In a graphic full of elements it is hard to recognize a specific one and to discriminate it from other objects (e.g. to find the exact position and/or boundaries). While searching for the element the observer is distracted and cannot pay full attention to your talk.

Forces

Adding and changing graphic elements on demand makes an illustration adaptive and flexible but can also mess up the screen or hide important information.

Turning graphic elements on and off allows selecting and filtering the information displayed on the screen but how can one element be selected as a special one? How to focus and highlight one element instead of treating all elements equally? How to ensure that only one object out of a group is activated and shows a special visual state?

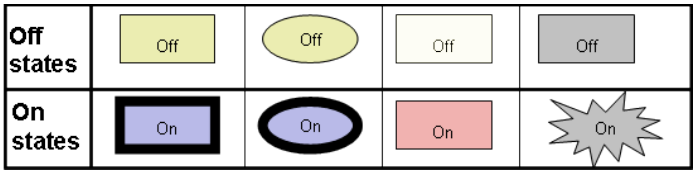
In some graphics one has to avoid that multiple elements are selected simultaneously but users may forget to deselect other elements. Deselecting an element may be inconvenient as it involves extra mouse moves. To deselect an element, the user has to search for the currently activated element on the screen but graphic elements may not provide explicit visual hints which element is currently activated.

Temporarily activating elements or groups of elements could be done by RADIO BUTTONS as well. But if the mouse pointer has to be available for other operations the activation and deactivation of elements must be triggered explicitly and not bind the mouse pointer. TOGGLE BUTTONS support such an explicit change of states but do not ensure that only one element is focussed at a time.

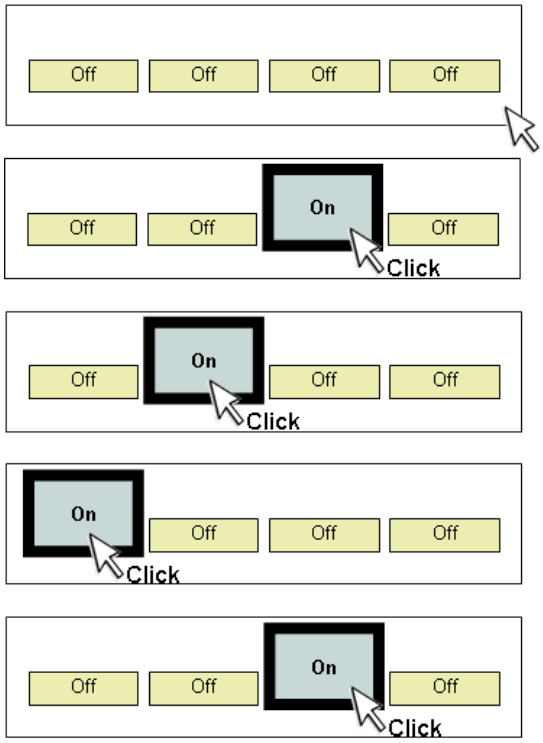
Solution

Allow the user to explicitly highlight or change an element by clicking at it. On activating the element make sure that all other related elements are automatically deactivated.

To highlight the elements of a group, define two different visual states for each element of the group. In other words, define ON and OFF states for each element by varying visual properties:

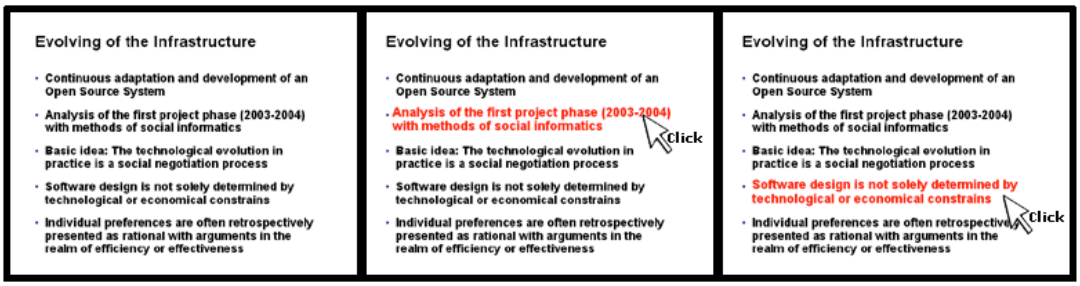


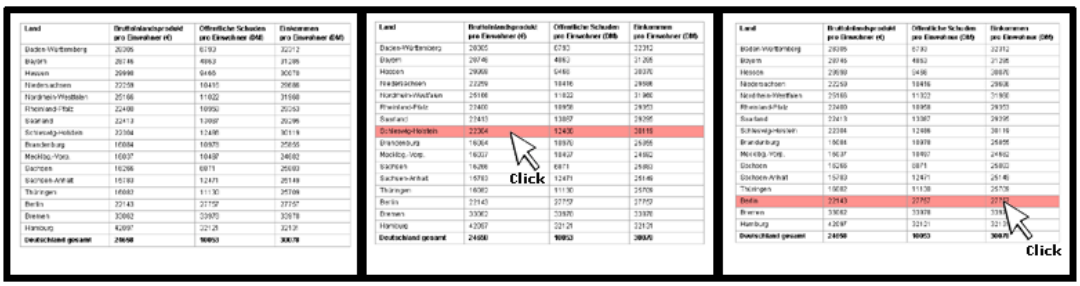
If the user clicks at any of the elements, the clicked element switches to its ON state and all other elements switch to their OFF states at the same time:



Details

The main purpose of a radio button is to select or activate only one element out of a group at the same time. One can select a single element to focus on it and direct the attention to that element. An element that is highlighted that way stands out and all other elements of the group fade out – either completely or partly. By directing the attention, one reduces the complexity of the graphic because the observer has not to care about all parts at the same time. Also, it is indicated directly which element is currently talked about. In particular, this can help in presentations. One can highlight the bullet item or the row of a table one currently talks about:





Whenever an illustration shows a situation in which only one element can be active concurrently, a radio button is very useful. For example, one can only listen to one radio station at a time (that's where the name "radio button" comes from), only perform one exercise at a time, or allocate scarce resources to only one object at a time. The next example shows how a radio button indicates who the next speaker in a meeting is:

By turning the last activated object automatically OFF, the user input is minimized and the shown information becomes not overloaded.

Once an element is selected to pop out, it remains permanently in ON state until another element of the group is activated. Thus, the mouse pointer is only needed once to activate an element and can be used for different tasks thereafter (in opposition to ROLLOVER BUTTONS which stress the mouse button for the time it activates an element).

If the user clicks at a button that is already in ON state there are two variations to react: the button may either remain in ON state (meaning there is always one of the elements activated) or it switches to OFF state (meaning that all elements are in OFF state).

At the beginning one of the buttons can be set to ON by default. Another option is to have all buttons set to OFF at the beginning. Note that the visual properties that change can be varied individually for each button. For example, one button could change its opacity while the other button changes its colour.

What can I do with this interaction form?

- Highlight or select one object of a group.
- Direct attention to a particular object.
- Show explicitly that for a group of objects only one can be active or activated concurrently.
- Show what to do next or who is in the row next.
- Indicate which object is currently focussed on and handled in a presentation.
- Reduce complexity by hiding or fading all information objects that are currently not needed.
- Focus on objects step by step in random access.

Rationale

A radio button selects one element out of a group and provides and implicit focus [Thi90]. It is taken care of that only one object is highlighted at a time. Thus, accidentally activating multiple objects is prohibited. This is important if the graphic is used to show that only one element should be focussed on or represents a special state. The activation state is automatically transferred between the buttons in the group. Highlighting or focussing an

object directs attention and clarifies which object is talked about. Radio buttons allow having all buttons simultaneously visible while assigning one special visual state to one of the buttons. Thus, information can be available at all times but it is assigned with different priorities.

Drawbacks

Each button can only represent two visual states. There is no explicit indicator which element is currently selected because the mouse pointer can move to other positions while the selected element remains in ON state. Thus, the graphic element has to provide its own visual hint that it is activated.

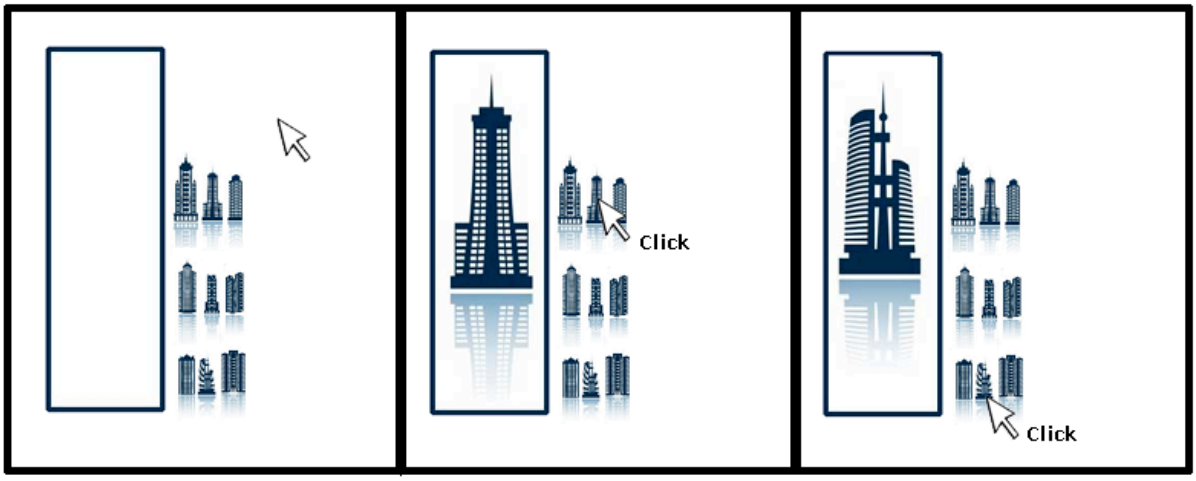
The graphic has to provide affordances to point out which areas are clickable in order to activate one of the radio buttons. These affordances may add an undesirable interference with the graphic itself. Radio buttons are often used to reduce the number of visual elements of a graphic. Hence, if new visual elements are required to activate the buttons, the benefits are nullified. For this reason, radio buttons work best if a base graphic is in use that offers implicitly such affordances, e.g. the bounds of a map or the spatial parts of objects or charts.

Information Display

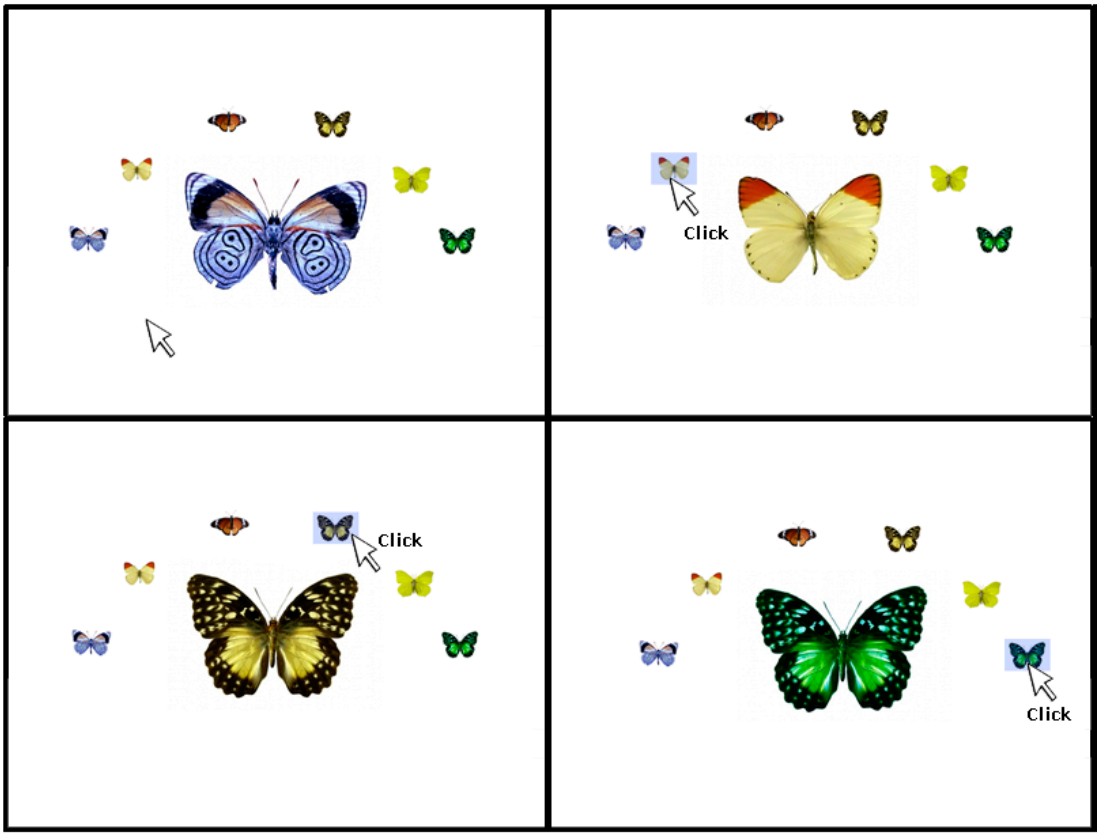
Provide a display to show additional information about other objects on the screen.

Examples

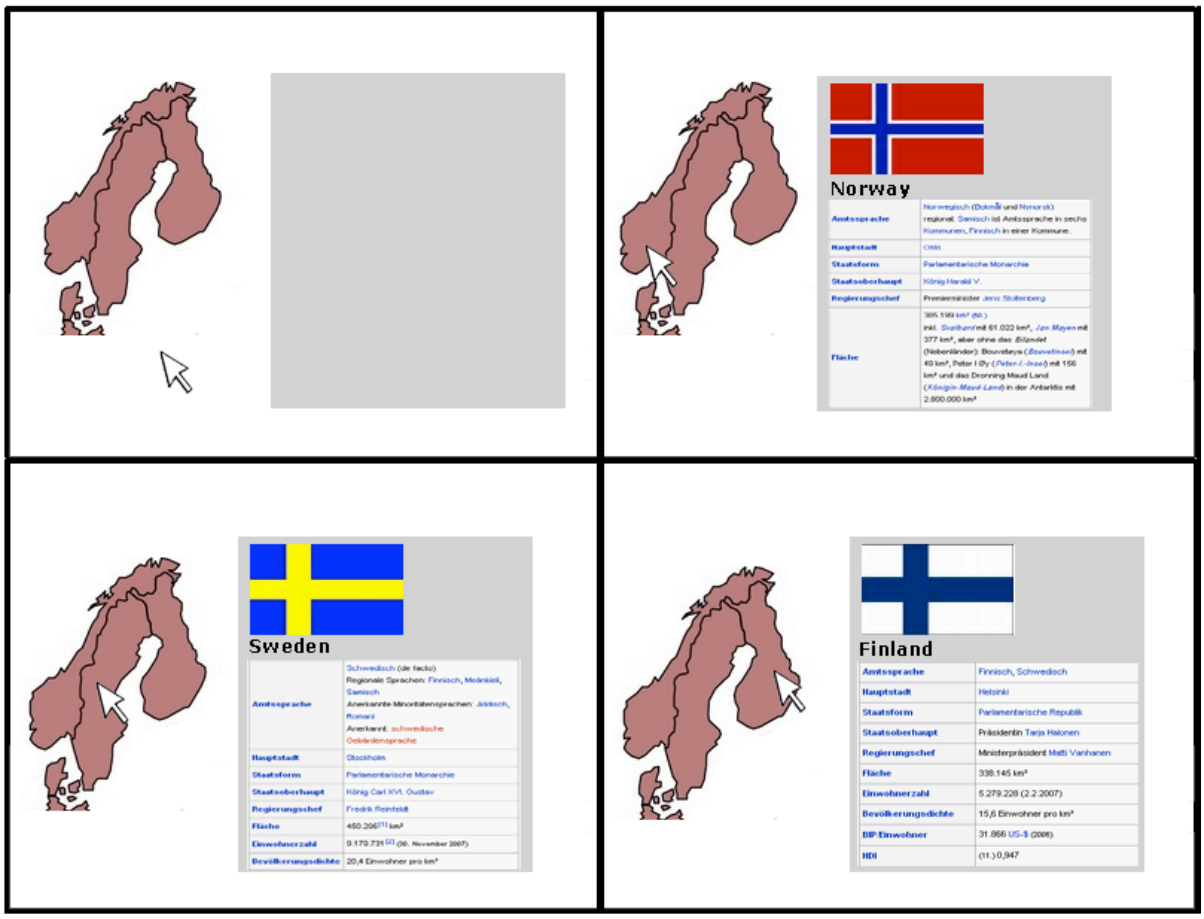
Select a skyscraper: The user can click at one of the skyscraper icons to get an enlarged illustration shown in the box.



Select a butterfly: The user can click at one of the small butterfly icons to replace the large butterfly illustration shown in the centre.



Info box: Clicking at one of the Scandinavian countries sets the content of the grey info box. The name of the country, the flag and statistical information are shown in the info box.



Context

For illustrative purposes you are using an information graphic in your classroom or in multimedia materials. For specific elements of the graphic you want to provide further information or look into the details. The base graphic should always be visible for reference and any additional information must not hide or replace the base graphic.

Problem

Overlaying the base graphic with additional information sometimes causes interfering with the content, in particular if a large text or image element is added. But placing additional information around the base graphic uses extra space on the screen. The available space is limited due to low screen resolutions. Reducing the base graphic in size decreases its quality and makes it harder to find orientation in the presented screen.

Forces

Adding information spatially close to an inspected object avoids split attention effects but overlay information interferes with the background and often hides important information. This is particularly true for larger pop-up frames.

If extra information is located close to an object, the user perceives it as a description of the objects. However, objects can also be used to add or set information that is only indirectly related to the object.

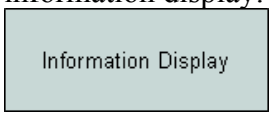
Integrating extra information directly into a base graphic implicitly relates it to contextual semantics but the new information may be independent or related to other contextual information as well.

Adding or changing information in a base graphic forces the user to struggle with the information while s/he may prefer to decide on her/his own when to access it.

Common buttons can change between two visual states but sometimes multiple state representations are required. A mechanism is needed to select one out of multiple states.

Solution

Use an element as a separate information display that can change its visual states to show information about different elements in a base graphic. Set a default visual state for the information display:

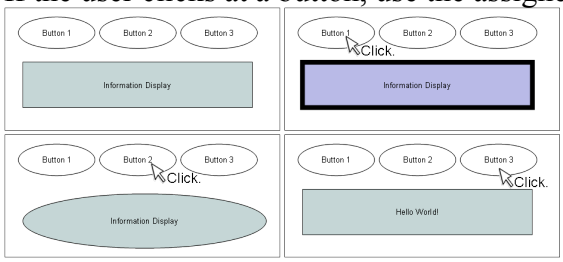


Overlay the base graphic with a set of button elements or hot area elements that can alter the visual state of the information display:

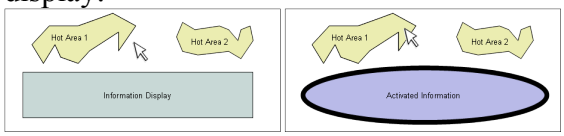
For each button/hot area assign specific visual property values to the information display:

Button	Default	Button 1	Button 2	Button 3
Display	Information Display	Information Display	Information Display	Hello World!

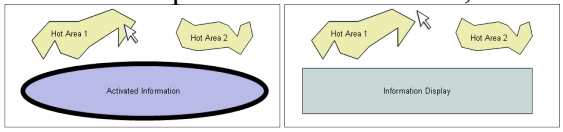
If the user clicks at a button, use the assigned visual state for the information display:



If the mouse pointer enters a hot area, use the assigned visual state for the information display:



If the mouse pointer exits a hot area, use the default visual state for the information display:



Details

An information display can change its content by clicking at a button element or by entering a hot area element in a base graphic. The information display is separated from the base graphic which activates the content of the information display. Thus, the content of the display does not overlay the original graphic and hide or interfere information. Showing extra information in a separate area is less disturbing. However, the larger spatial distance between the focussed area in the base graphic and the information given in the display decreases the overall coherence. Locating the eyes on either the objects in the base graphic or the objects in the information display costs cognitive resources. Using hot areas and mouse rollover to set the information display supports reorientation as the mouse points directly to the area of interest in the base graphic.

This orientation may get lost when using a button object, as the user can move the mouse independently after clicking the button. On the other hand, information displays are less transient if buttons are used.

The base graphic can be complex and the information display can show details or further information about elements of the base graphic. Another option is to use the base graphic as a menu. In this case, the base graphic's elements are menu buttons that can select the content shown in the information display.

Rather than having only two states (like TOGGLE BUTTONS), an information display has multiple visual states. The number of states is set by the number of buttons or hot areas that can set a specific state for the information display. The first shown state can be either an additional default state, or it can be one of the regular button (or hot area) states preset.

The information display is perceived as a passive element on the screen. Therefore, it cannot be involved as a button or hot area itself. If you want to provide a method to reset the information display to its initial default state you should consider an additional button rather than using a click on the display to reset it.

While the separation of the base graphic and the information display increases the spatial distance and makes the complete graphic less coherent, it becomes more clearly what is the base information and what information is additional or more detailed and zoomed in. Not only does the display avoid visual overlapping of information, it also offers a large frame for information if needed. Thus, explaining elements of the base graphic can be done more elaborately than in small pop windows or text bubbles.

What can I do with this interaction form?

- Show a status bar.
- Provide text information for objects on the screen.
- Show details (e.g. a zoom view) for one of the objects on the screen.
- Use the button/hot area objects as menu items to select content.
- Explain components of a larger structure or object.
- Provide information about areas of a map.

Rationale

The main purpose of the information display is to provide details on demand [Shn87]. The base graphic is a means to select which information should be displayed. In that sense it is a visual navigation menu which the user can use to navigate to information nodes. The content of the information nodes is shown in the information display. Menu bars and content frames are a special case of an information display. An information display allows zooming into the details while having the overview picture shown at the same time [War04]. Relations between the big picture and its details are better perceived. Information displays can also be used to show small multiplies [Tuf90] and having one or two of the thumbnails enlarged (e.g. for comparison).

Drawbacks

Information can only be given for one element at a time. Separating labels and images is a cause for split attention effects. The eye's focus has to shift between the information given in the base graphic and the information display. In rollover information displays, the mouse pointer can help for orientation. However, using the mouse pointer for activation disables the mouse for any other activation. Clickable information displays release the mouse for other tasks but it is harder to refocus on the activated area in the base graphic. An interesting option to help this problem is to combine an information display with RADIO BUTTONS that highlight the area in the base graphic which set the display.

An information display never alters the triggering element itself but sets the state for another display. The display is usually at a distant location and has to be perceived as an active element on the screen. If the display is small (e.g. a status bar at the bottom) the user may miss the added information or the change of content.

Acknowledgement

Special thanks to Tim Wellhausen who has shepherded this paper and shared his experience with the authors. We are especially thankful that Tim volunteered to shepherd an extra paper and jumped in to backup. He was a great support and helped to improve the paper! We would also like to thank all workshop members at the EuroPLoP 2008 in Irsee who have given very valuable feedback.

References

[Ba97] Bayle, Elisabeth et al. : Putting It All Together : Towards a Pattern Language for Interaction Design, Summary Report of the CHI'97 Workshop

[Bor01] Borchers, Jan: A Pattern Approach to Interaction Design; John Wiley & Sons, 2001

[CS91] Chandler, P., and Sweller, J.: Cognitive Load Theory and the Format of Instruction. *Cognition and Instruction* (pp. 293-332) 8 (4). 1991.

[CM02] Clark, R. C., & Mayer, R. E.: E-Learning and the science of instruction. Proven Guidelines for consumers and designers of multimedia learning. San Francisco: Jossey-Bass/Pfeiffer. 2002.

[DLH04] Duynie, Douglas K. van; Landay, James A.; Hong, Jason I.: The Design of Sites, Addison-Wesley, 2004

[Gib79] Gibson, J. J.: The Ecological Approach to Visual Perception. Boston: Houghton Mifflin, 1979.

[Gol89] Goldstein, E. Bruce. Sensation and Perception. Belmont, Calif: Wadsworth Pub. Co, 1989.

[GLC01] Granlund, Asa; Lafrenière, Daniel ; Carr, Daniel A.: A Pattern-Supported Approach to the User Interface Design Process, Proceedings of HCI International 2001, 9th International Conference on Human-Computer Interaction, 2001, New Orleans
[KU09] Kohls, C. and Uttecht, J. G. (in press). Lessons learnt in mining and writing design patterns for educational interactive graphics. Computers in Human Behavior.
[KW06] Kohls, C., Windbrake, T. Towards a Pattern Language for Interactive Information Graphics. Pattern Languages of Programming Design 2006. Portland, Oregon: Hillside Group. URL: http://hillside.net/plop/2006/accepted_papers.htm.
[Mah06] Mahemoff, M: Ajax Design Patterns. Creating Web 2.0 Sites with Programming and Usability Patterns. O'Reilly Media, Sebastopol, 2006
[MJ98] Mahemoff, M.; Jonston, L.: Principles for usability-oriented pattern language, OZCHI '98 Proceedings, Adelaide, Australia, S. 132-139
[May01] Mayer, R. E.: Multimedia Learning. Cambridge: Cambridge University Press. 2001.
[MLC05] Malone, E.; Leacock, M.; Wheeler, C.: Implementing a Pattern Library in the Real World: Yahoo! Case Study. http://www.leacock.com/patterns/ (accessed 01.04.06)
[MM99] Moreno, R.; Mayer, R.E.: Cognitive Principles of Multimedia Learning: The Role of Modality and Contiguity, Journal of Educational Psychology, 91, p. 358-368, 1999
[ND05] Niegemann, Helmut M.; Domagk, S: ELEN project Evaluation Report, Report of Work package 5. E-LEN project: a network of e-learning centres; http://www2.tisip.no/E-LEN/documents/ELEN-Deliverables/Evaluation_Report_E_LEN.pdf (accessed 29.03.06)
[PPP] The Pedagogical Patterns Project, http://www.pedagogicalpatterns.org/
[Rie90] Rieber, L.P.: Computers, graphics, & learning. Englewood Cliffs, NJ: Prentice Hall. 1990
[SM00] Schumann, H.; Müller, W.: Visualisierung. Grundlagen und allgemeine Methoden, Springer, Berlin, 2000
[Sc05] Schmitt, Silke; Schreiner, Martin; Timmesfeld, Fel; Vucica, Martina; Wallach, Dieter: PatternCube.com: Ein webbasiertes Repository für User Interface Design Patterns. In: Hassenzahl M.; Peissner, M. (Hrsg.): Usability Professionals 2005
[Shn87] Shneiderman, B.: Designing the User Interface: Strategies for Effective Human-Computer Interaction. Reading, Mass: Addison-Wesley, 1987.
[Thi90] Thimbleby, H.: User Interface Design. ACM Press frontier series. New York, N.Y.: ACM Press, 1990.
[Tid05] Tidwell, Jenifer: Designing Interfaces, O'Reilly, Sebastopol, 2005
[Tuf90] Tufte, E. R.: Envisioning Information. Cheshire, Conn.: Graphics Press, 1990.
[VW04] Vogel, R; Wipperamnn S.: Dokumentation didaktischen Wissens in der Hochschule Didaktische Design Patterns als eine Form des Best-Practice-Sharing im Bereich von IKT in der Hochschullehre, Wissenschaftsforschung Jahrbuch 2004, Berlin. 2005
[War04] Ware, C: Information Visualization – Perception for Design. Morgan Kaufmann Publishers, San Francisco., 2004
[Wel05] Wellhausen, T. User Interface Design for Searching - A Pattern Language. http://tim-wellhausen.de/papers/UIForSearching.pdf (accessed 19.06.2008)
[WV03] van Welie, M.; Veer, van der Gerrit, C.: Pattern Languages in Interaction Design: Structure and Organization, Interact 2003

Patterns for Supervising Thesis Projects

Axel Schmolitzky

University of Hamburg, Germany
Vogt-Kölln-Str. 30
D-22527 Hamburg
+49.40.42883 2302

schmolitzky@acm.org

Till Schümmer

FernUniversität Hagen, Germany
Universitätsstr. 1
D-58084 Hagen
+49.2331 987 4371

till.schuemmer@fernuni-hagen.de

Abstract: Thesis projects are a challenging task for students as well as their supervisors. In most cases, students have not managed such large projects before. Many supervisors are good researchers, but have not received training in pedagogy and project management. This means that students as well as supervisors often lack best practices in managing thesis projects. This paper fills this gap by providing a set of best practices for the supervisor that may help to better structure and focus the collaboration between student and supervisor so that the thesis runs smoothly, thus enabling students to succeed.

1. Introduction

The following patterns describe best practices for supervisors of students' thesis projects. Students have to write such a thesis at the end of their bachelor, masters, or diploma program (the "Diplom" was the most common degree in higher education in Germany until the mandatory change to Bachelor/Master degrees). These projects are typically long-term interactions between the supervisor and the student that last between 3 and 12 months, sometimes even longer.

Typically, after they have agreed on a topic with their supervisor, students can take some time to familiarize with it and prepare for the official work term on their thesis. Ideally, they use this time to reach a good understanding of their subject and create a realistic plan for their practical and theoretical work. Quite often this precedes the time officially allotted to the thesis.

One common problem is that students and supervisor do not meet on a daily basis. Even if the students participate in a research group, typically the full-time researcher cannot interact with them all the time. In addition, many students have a part-time or even full-time job, so that the thesis work has to take place in the evenings or on weekends. This is especially the case at distance teaching universities like the FernUniversität in Hagen, Germany. Thesis projects are thus an example for a blended learning setting: Co-located synchronous phases of interaction interweave with phases where students work at home, following their individual schedules and preferences for the work setting.

Proceedings of the 13th European Conference on Pattern Languages of Programs (EuroPLoP 2008), edited by Till Schümmer and Allan Kelly, ISSN 1613-0073 <issn-1613-0073.html>.

Unfortunately, the freedom of working independently from the supervisor can lead to phases where the thesis moves out of the student's focus. Especially in distance teaching universities, we can observe frequent drop outs of students due to private matters or a high workload at the student's workplace. Students abort the thesis project before they have actually started the official part of the project. In our experience, a closer interaction between supervisor and student starting on the day of the first encounter helps to reduce the number of drop outs and keeps the students focused on their theses.

The patterns introduced in this pattern language can guide supervisors in such a close interaction. We started to write down these patterns for various reasons:

- After several years of experience in supervising students during their thesis work, certain patterns were becoming too obvious for us to be ignored.
- Some best practices we apply today would have saved us a lot of time in our early years as supervisors if we had been aware of them or been able to use them. And we regretfully notice that novice colleagues tend to make the same mistakes as we did when we first supervised thesis projects.
- Initially, we thought that there were already patterns for this subject. However, we were not able to spot articles discussing this issue.
- Some books on the subject, such as [6], give good advice, but do not cater to new developments such as agile methodologies. As many students' thesis projects develop rather into an expedition than into the manufacturing of a product, agile practices helped us a lot in supervising thesis projects.
- The integration of supporting computer technology (such as wikis, email, repositories) into the supervising process has changed further and improved the processes for us. We are not aware of any literature that captures how technology support interrelates with the social practices of thesis supervision.
- Finally, we think that a compact description in pattern form has more potential to be widely noticed than any book on the subject, even if it is as concise as [6].

Although these patterns are targeted at thesis supervisors, we think that they can also help students working on their final theses. The patterns can act as a guideline for both parties and help to make the expectations of supervisors and students more explicit. This has the advantage that students can adapt to the way how supervisors think that thesis projects should work. The patterns draw the picture of an ideal student who is responsive and makes her process transparent. But they also describe how an ideal supervisor should take care of the student. Thus, the patterns require a high level of discipline on both sides, and supervisors and students should be aware of the fact that both parties might fail to implement some of the patterns. In our experience this is not critical, as long as there is an open and honest communication culture. Neither students nor supervisors should close their eyes, but speak up when observing situations where one side fails to play their role.

We explicitly exclude supervising Ph.D. theses from our patterns for two reasons:

- As a Ph.D. student has to work much more independently, less guidance

should typically be necessary.

- None of us has enough experience in supervising Ph.D. theses.

This does not mean that some of these patterns cannot be used in Ph. D. supervision. However, we have no experience with the application of these patterns in Ph. D. theses. If you are a Ph. D. thesis supervisor, you may have a look at [3].

As noted before, we consider thesis projects to be instances of a blended learning setting. Consequently, the patterns of this pattern language are written as socio-technical patterns. Each pattern starts with a context description and a problem statement that summarizes the main reasons why the pattern should be used. After that, we list a set of forces that were considered in the pattern. We understand these forces as conflicting requirements in the interaction between supervisors and students. The goal of the pattern should be to change the socio-technical setting of the process in a way that the forces are less conflicting. In an ideal situation, the solution would remove the mentioned conflict between the forces.

The solution names social interaction between student and supervisor. After this, we discuss the design and use of technology that can support the social interaction between supervisor and student. In most cases, it is sufficient to employ standard technology including communication technology (telephone, e-mail or instant messaging systems) and shared information spaces like wikis or shared file systems (e.g., BSCW or Google Docs). In some cases, a tighter integration of technology can be needed, which is described as a design guideline for technology designers. Note that all patterns can also be implemented without any technology support.

2. The Pattern Language

This paper contains the following patterns:

2.1 FIRST ENCOUNTER: Your first meeting creates the basis for a trusting and efficient work relationship. Be thorough in defining the *formal* context and leave the details of the subject of the thesis for later.

2.2 PROJECT HEARTBEAT: Request status updates on a regular basis to ensure that the student is still participating in the project.

2.3 AGILE EXPOSÉ: Make the student write an exposé for the thesis and make the student update it based on your feedback until the task is well-defined for both the student and you.

2.4 EARLY OUTLINE: Let the student write and maintain an outline of the final thesis as soon as the scope of the project is well-defined.

2.5 STUDENT-MANAGED SCHEDULE: Ask the student to create a project schedule and ensure that the student updates the schedule when the work progress deviates from it.

2.6 DIARY: Propose that the student writes daily notes on her/his progress in a diary, so that ideas and decisions stay persistent throughout the project.

2.7 ADVISED LITERATURE RESEARCH: Let the student collect and reflect on literature in an interaction with you.

2.8 TEST THE WATERS: Identify the student's strengths and weaknesses by assigning tasks on a small scale. Upon completion, ask for the time needed, so that you learn about this student in particular and about students' performance in general.

2.9 EXPRESSIVE STUDENT: Ask the student to present the core of her/his research at different stages of the thesis project and on different levels of granularity also to other people than yourself.

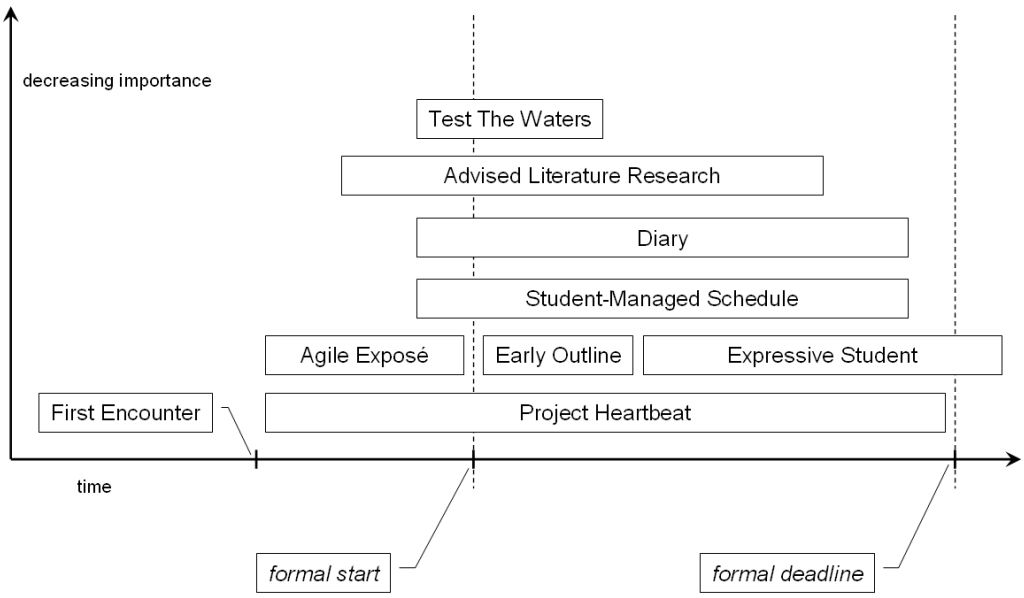


Figure 1: The patterns on a timeline

Figure 1 shows the patterns of this pattern language on a timeline. Any thesis project has a formal starting point and a formal deadline. These dates are typically enforced by official regulations of the educational institution. In the figure, the patterns are shown in relation to these points in time. For example, according to the figure, AGILE EXPOSÉ should be fully applied before the project is formally started. DIARY can be started at some point before the formal start and EXPRESSIVE STUDENT can be applied before and even after the formal deadline.

When selecting patterns for a concrete thesis project, we suggest you first select the patterns closer to the timeline since these patterns are more important than those shown at the top of the figure.

2.1 FIRST ENCOUNTER

Context: A student is looking for a subject for her/his thesis. You fulfill the formal prerequisites for being a supervisor. The student asks you if you would be willing to supervise her/his thesis.

Problem: You need to find out whether cooperation with the student can work out. If you do not clarify expectations upfront, there is too much room for misunderstandings and conflict.

Forces:

- You might not know the student very well, maybe only from one or two previous courses. Especially you might not know much about the student's abilities.
- The student might not know you too well, either. S/he might not know what exactly you expect from her/him.
- Procedures might not be clear if no formal framework for theses is applied in your institution.
- The topic should be tailored to the student's preferences and capabilities.
- Your additional workload of yet another thesis supervision should also pay off for yourself.

Solution: Meet the student to tell her/him how you handle thesis projects. Give as much information about your way of supervising as possible. Pass on this pattern language. Ask the student about her/his personal situation, including:

- What degree is the student aiming for (Bachelor, Diploma, Master's Degree)?
- In which program (e.g., Computer Science, IS) is the student aiming for a degree?
- When is the student planning to start working on the thesis?
- Are there any external formal or hard deadlines, e.g., exams, expecting a child, long-planned holidays?
- Is the student planning to work on the thesis full-time, part-time or at nights?
- What kind of degree is the student aiming for? Is s/he ambitious or just looking for some final thesis?
- What preferences does the student have with regard to helpful skills, such as programming, theory building, interviewing, or writing?

After the student got an impression of the formal context, ask how much time per week (in days) the student is willing to spend on the thesis. Based on the answer, calculate the earliest date you can think of for finishing the desired type of thesis (e.g., the student is willing to work three days a week. For a half-year full-time diploma thesis this means that the work will have to last for at least nine months!). Try to fit this with your personal context (e.g., you might prefer to get the final thesis in your semester break) and fix a deadline with the student.

Even though this might not be the actual formal deadline, having the time frame fixed gives both the student and you a good base for planning.

Technology Support: When agreeing on a date for the first encounter, you can ask the student to send you a short CV to learn more about her/his background. Together with a confirmation of the appointment, you can send out an agenda as well as a link to your personal thesis guidelines. These guidelines are a public document in which the general rules and assumptions for the supervision of thesis projects are shown. You may even consider passing this pattern language to the student.

In cases where you use a shared workspace system for supporting the interaction between you and the student, you should prepare the FIRST ENCOUNTER by cloning a template workspace that already contains information about structuring a thesis project.

Discussion: The first encounter helps to reach a mutual understanding of each other's goals and expectations. The questions stated in the solution can serve as a check list and ensure that all important information is exchanged.

Although it can work in some cases, we advise against starting a thesis project without an initial face-to-face meeting, even in a fully distributed setting like the FernUniversität in Hagen. The meeting helps to create a mutual understanding as well as an impression of the student (and the student will get an impression of you). See also "Face to Face before Working Remotely" in [5] where the authors recommend to have a face-to-face phase in the early days of a development project before individual sub-teams start working in remote locations. If you cannot arrange a face-to-face meeting, you should have the best possible meeting infrastructure in place for this meeting including at least high quality video and audio connections. In an ideal setting, you would also use shared whiteboards for creating hand-drawn figures and application sharing systems for looking at example systems together. But even then, we do not recommend a remote FIRST ENCOUNTER.

Related Patterns:

- 2.3 AGILE EXPOSÉ: Typically, the first task for the student after the initial meeting is to create a problem definition that captures her/his understanding of the thesis' goals.
- 2.5 STUDENT-MANAGED SCHEDULE: You can start discussing the cornerstones of a schedule during the FIRST ENCOUNTER.

2.2 PROJECT HEARTBEAT

Context: The student is working on the thesis.

Problem: To successfully finish the thesis project, the student has to keep up the pace. But it is hard to detect changes in the student's pace that make appropriate support and coaching necessary. The thesis eventually runs completely out of schedule.

Forces:

- The student is not co-located with you (e.g., is not working in your research group), so you have no casual or regular contact.
- Many external forces (workload, family-related issues) can hinder the student's progress in such a way that major rescheduling becomes necessary.
- The project can lose its momentum because the student needs more pressure from your side.
- The thesis might not be part of your own research agenda, so you have no intrinsic interest in its progress; without active input from the student the thesis shifts out of your focus.

Solution: Propose a social contract that forces the student to report the project's progress at least every 14 days. Ask the students to summarize the work and their insights since the last report. Whenever a report is overdue, remind the students of the violation of the social contract and propose a meeting where the future of the thesis work is discussed.

Technology Support: The student sends you an e-mail reporting on the latest progress. You store the latest mail in a thesis folder and mark this mail for tracking after 14 days. You frequently check the mail folder and contact those students who have violated the social contract.

An integrated system can even further improve the social process outlined by the pattern. The system can keep track of the last activity summary and prompt the students to update their activity summary in the agreed intervals. Both the individual student and you can see the date of the last report. In addition, you see an overview of due reports for all of your students.

Discussion: Project heartbeat is closely related to the ALIVENESS INDICATOR presented in [10]. To our experience, the 14 days period has shown to be an effective time span for not losing the mutual awareness. If the DIARY pattern is applied, the regular entries can be considered to be PROJECT HEARTBEAT. In such cases, an integrated groupware solution would keep track of the dates of the latest diary entries.

Related Patterns:

- 2.6 DIARY: The DIARY provides more information on the progress of the thesis. On the other hand, it requires additional efforts from the student. The PROJECT HEARTBEAT can thus be considered an automated and lightweight version of the DIARY. We consider PROJECT HEARTBEAT to be mandatory and DIARY to be desirable.

2.3 AGILE EXPOSÉ

Context: The student has expressed interest in a specific problem area. You are interested in specific results, like the creation of a software component, the analysis of usage data, or a literature overview of a specific field. Such results are valuable to you as a researcher and help you to gain new insights, e.g., in the context of your larger research agenda. The problem statement for the task has been discussed during the FIRST ENCOUNTER.

Problem: The student and you have different visions and goals for the result of the thesis project. If the goals are too different, the student will create a solution that does not meet the supervisor's expectations. This can result in a poor grade and unusable results.

Forces:

- You have identified an interesting problem, but you do not have an idea for a solution to the problem yet. Note that solution ideas can evolve out of discussions.
- You have a clear vision for the thesis, but you have failed to communicate it to the student during the FIRST ENCOUNTER (1).
- The more the student delves into the topic, the more aspects will s/he be able to contribute to the problem definition.
- The task should be challenging and scientifically relevant.
- The task should be tailored to the student's preferences and capabilities.
- You have to guarantee that the task is appropriate for a thesis.

Solution: Ask the student to summarize the plan for her/his thesis in her/his own words by writing an *exposé*. An exposé is a text of 2 to 6 pages length, describing the context, the problem, the approach to a solution (if appropriate) and a rough schedule for the actual work on the thesis. You comment on the exposé and ask the student to rewrite it until it embodies a shared understanding of the task. This process can take several iterations and thus several weeks. It is the *collective responsibility* of the student and you to reach such a written 'agreement' both are satisfied and comfortable with.

Technology Support: The supervisor creates an empty wiki page for the task description. At the end of the meeting, s/he asks the student to summarize her/his understanding of the task and to send the complete summary to the supervisor. Upon receiving the mail, the supervisor edits the wiki page and highlights points where s/he has a different understanding of the task. This is repeated until the supervisor sees no more differences.

More task oriented groupware applications can improve the coordination between student and supervisor. Typically, such systems provide explicit FLOOR CONTROL [10] for the document. After the student has finished the task summary, s/he passes the floor on to the supervisor who will get informed immediately. The supervisor can use SHARED ANNOTATIONS [10] for pointing out differences in understanding and pass the floor back to the student.

Discussion: An exposé should not be seen as a small version of the final thesis (sometimes students write their thesis text based on their exposé).

Instead, an exposé describes upfront what should be done, with a focus on the process, including a timeline with a description and an estimation of the necessary subtasks. The structure of the final thesis text can be very different from this (and typically is). After the student and you have agreed on a stable exposé, it does not need to be changed again. It can serve as a contract that defines the scope of the work.

Related Patterns:

- 2.9 EXPRESSIVE STUDENT: It is important that the exposé is written by the student. But while EXPRESSIVE STUDENT aims at the student's ability to convey the core idea of the thesis project to other people than you, this pattern focuses on the relationship and the mutual understanding between the student and you (the supervisor).
- 2.8 TEST THE WATERS: When writing the AGILE EXPOSÉ, you learn more about the student's writing skills. For the student it is a first test of writing a text that you must agree with.

2.4 EARLY OUTLINE

Context: The student has started to work on the thesis.

Problem: Students have difficulties to start the writing process, especially when they see a blank screen or an empty sheet of paper. They have very limited experience on structuring their ideas in a way that is suitable for a scientific thesis.

Forces:

- Students typically underestimate the time needed for writing the final thesis text.
- Often students have no prior experience with writing a larger document.

Solution: Ensure that the student creates an outline of the thesis directly after the scope of the thesis has been defined. Provide a prototypical thesis structure and ask the student to adapt the structure to her/his specific problem. Ask the students to add one paragraph explaining the intended content for each section in the outline.

A prototypical outline might look like this:

1. Introduction

This section motivates the problem that is solved in the thesis. It provides first explanations why the thesis is worth reading and explains its structure.

2. Problem Analysis

The goal of this section is to explain the background of the investigated problem. It explains why it is a problem and deducts a set of requirements that need to be fulfilled for finding an optimal solution. It may also point out conflicting requirements and request that the conflict is resolved. It may make sense to conclude this section with a table showing all requirements.

3. Existing / Other / Related Approaches

This section should contain a list of related approaches or solutions that could be applied for solving the problem and satisfying the requirements. Each approach should be summarized and discussed with respect to the requirements. At the end of this section, you should provide a summary of deficits of the state of the art.

4. Approach of the Thesis

Explain the concepts of your approach and show how you address the requirements. In cases where you build something that users should use (e.g., interactive software), it can be good to show how your solution is used.

5. Solution Details

Explain details of the solution. The description should be detailed enough to allow a peer researcher or practitioner to re-implement the solution. This section may also contain studies on how the solution has been used.

6. Evaluation

Reflect on the effectiveness of the solution. Show evidence, if possible. Evaluation can be done quantitatively or qualitatively, depending on the solution and the context.

7. Conclusions

The final section should serve three purposes: (1) To summarize the approach; (2) To compare the approach to the state of the art; and (3) To point at top directions of future research and development.

A. References

The outline should be a living document that should be updated regularly during the thesis project.

Technology Support: Create a wiki page for a prototypical outline. Copy this page after the student has created the problem statement and send it to the student with a request for adapting the outline. Periodically ask the student if the outline is still aligned with the current thesis.

Discussion: Sharing the thesis outline may result in a new pair of conflicting forces: The outline should provide a high-level overview of the thesis, while the thesis itself contains the real content. Changes in the outline affect the document and vice versa. All current text processing systems thus automate the process of outline creation. However, if only one part is modeled as a shared document, this synchronization may be more difficult.

Related Patterns:

- 2.5 STUDENT-MANAGED SCHEDULE: The schedule explains how and in which order the different parts of the outline will be filled. Both schedule and outline help to structure the student's work better.
- 2.3 AGILE EXPOSÉ: While the exposé describes what *should be* done, together with a first version of a schedule, the outline is a mini-version of the final thesis text und should describe what *is being* done.

2.5 STUDENT-MANAGED SCHEDULE

Context: The student and you have agreed on a problem statement and the student is about to start working on the thesis.

Problem: Students are independently managing their time. However, they often lack experience in planning a long-term project such as a thesis. If this management is done in an unstructured way, students overlook critical deadlines. **As a result, in most cases the final phase of a thesis project is accompanied by a high level of stress and may result in a quality decrease.**

Forces:

- It is hard to predict the future, especially in a research project. Examples of common pitfalls are that
  - the student underestimates the time needed for writing the final text;
  - personal problems or the student's job suddenly require more time than expected;
- in order to finish the project on time, the student needs a plan;
- the student is not used to making and following a plan;
- the student is not aware of (potential) upcoming problems;
- you as a supervisor are not aware of problems in the student's progress.
- Consequently, you fail to intervene or help the student when help is required.
- The student is not happy with the progress but fears to discuss problems with you since this might lower the final grade.

Solution: Ask the student to create and maintain a schedule for the thesis project and ensure that s/he discusses it with you. Both parties agree on a set of milestones where the student presents intermediate results to you. Think about the deliverables that have to be completed at the end of the thesis project and estimate roughly how long it would take to finish each deliverable. Remain on a coarse level of detail (e.g., tasks lasting for approx. one week). You renegotiate milestones if the student was unable to complete the required steps for a milestone. The schedule should be updated regularly.

Technology Support: You create a skeleton wiki page that includes the typical milestones for the thesis project. Before the student starts working on her/his thesis, s/he adapts the schedule to her/his needs and fixes dates and content for the milestones. You approve the schedule, e.g., by e-mail. Shortly before a milestone, the student informs the supervisor by mail about the current status of the project and arranges a presentation date for the milestone. Schedule updates are also negotiated by e-mail. Finished tasks are marked in the schedule wiki page.

You can ease the process of schedule creation and maintenance by integrating scheduling support in the e-learning system. Instead of thinking about concrete dates, the student estimates the required time for each task and defines the sequence of tasks. Afterwards the system creates a schedule that is visible to

both the supervisor and the student. Students are informed about approaching deadlines, and the supervisor is reminded of missed deadlines. This ensures that there is a high awareness of tasks that are overdue. When all tasks for a specific milestone have been done, the system automatically arranges a review meeting where you discuss the milestone.

Discussion: The first draft of a schedule can be taken from the AGILE EXPOSÉ.

The schedule defines criteria by which the student's progress can be evaluated. In this context, it can serve as the source of an INSTRUCTOR EVALUATION [7], a pattern that points out that the instructor should comment on the student's achievements. The main deficit of the INSTRUCTOR EVALUATION pattern is that it does not explicitly focus on the underlying social interaction. Several systems support project management in a similar way. However, most e-learning environments do not support task planning.

The student should not add too much detail to the schedule. This is the reason why we would not suggest the use of project-management systems such as MS-Project®. These systems tend to create a vision of safety although the research project as such still has a high level of uncertainty.

Related Patterns:

- 2.6 DIARY: The schedule should be reflected in the DIARY as soon as the plan is realized.
- 2.2 PROJECT HEARTBEAT: An alternative way of tracking the student's progress is to let her/him send regular messages to you. If these are the only record, the student should keep these messages as a DIARY equivalent.

2.6 DIARY

Context: The student is working on the thesis. The thesis involves design, experimentation and tests with different design alternatives.

Problem: The thesis project requires a long research and learning process. The student explores the state of the art, creates hypotheses and experiments to verify the hypotheses. The deeper the student delves into the work, the less reflection takes place. **Important insights and ideas may thus get lost during the project. In addition, in many cases the supervisor learns too late about problems and thus is unable to provide suggestions for improvements at the right time.**

Forces:

- Good ideas and new insights materialize during a thesis project, and not all of them can be implemented.
- The student forgets ideas and insights that s/he has not written down.

Solution: Ask the student to create a diary that documents the thesis project activities. The diary serves as a knowledge repository for all thoughts and insights, so that they will not be lost when the final thesis writing takes place. The diary or excerpts of it should be shared between you and the student at least. Frequently read the diary and see if the student ventures in the wrong direction. If needed, offer help, so that the student gets back on track.

Technology Support: The easiest way to implement a diary is to write it as a shared wiki page. However, student and supervisor have to agree on visibility levels which not all wikis support. In cases where privacy is an issue, the diary can also be created as a restricted wiki.

Further integration can link the diary writing activities to the student's workplace: Students log into the system when they start working for their thesis. Before beginning the work, they summarize their plan for the day. In the process of logging out, the system prompts them for a sentence telling what they have achieved this day. The summary is stored in the diary system which allows the student and the supervisor to browse all daily summaries of the thesis project. If there are unsolved problems, the student can mark these as action items for the next working session. Note that the system should allow students to mark entries as private, so that the supervisor cannot see these entries.

Discussion: Derntl [7] also describes a DIARY pattern. Due to the pattern structure used by Derntl, the problem is not clearly stated. In addition, the staged solution description makes it easier to apply the DIARY pattern in different e-learning systems. BLOGs are often used to support the collaborative creation of a diary in e-learning contexts. Moodle, e.g., offers students and teachers to co-construct a so-called journal that fills the role of the DIARY.

It can also be helpful for you to keep your own diary of the meetings with the student (a *supervisor diary*). As you typically supervise more than one student, this will help you to remember what has happened so far. This is especially useful if the student is not applying this pattern or if you have no access to the student's diary.

Related Patterns:

- 2.2 Project Heartbeat also suggests to provide periodic summaries of the progress made. As said before, a DIARY can replace the PROJECT HEARTBEAT under certain circumstances, but without a DIARY there should a least be an application of PROJECT HEARTBEAT.
- 2.5 STUDENT-MANAGED SCHEDULE: Whenever the student enters a note regarding a finished task in the DIARY, s/he should re-estimate how this helped her/him to finish schedule tasks.
- 2.7 ADVISED LITERATURE RESEARCH is an alternative for documenting and exchanging insights from literature studies.

2.7 ADVISED LITERATURE RESEARCH

Context: The student is working on the thesis.

Problem: Students need to evaluate research literature in order to relate their ideas to the state of the art. But in their previous studies, students were rarely confronted with research literature. Instead, they received pedagogically enhanced material that clearly stated questions, methods and results. **Without a solid base of references, the thesis might not be scientifically sound enough and thus you would have to give it a poor grade.**

Forces:

- Other people's results can be very inspiring and helpful for the thesis project.
- Working with literature is not as interesting as building a design artifact, such as a running software system.
- Students, especially in engineering disciplines, tend to think that literature references are just a formal detail belonging only in the final thesis text.
- Sometimes it is not easy to find relevant literature for a specific topic.
- The students expect you to provide them with relevant references.

Solution: Ask the student to fill a literature pool. Let her/him search, summarize and comment the literature. Make sure you obtain regular access to the literature pool and comment on the student's summaries. This is especially necessary if the student frequently uses "unsound" sources, such as "XX in 21 Days for Dummies" or Wikipedia articles authored by hobbyists that have not yet been reviewed by experts. Sometimes it is also helpful to get a second opinion from a colleague on the literature pool for the specific problem.

Technology Support: Use a wiki to manage the literature summaries. In cases where the wiki supports page templates, you should create a template that contains all required fields for the literature summary as well as the bibliographic data. After the student created a literature summary page, s/he shall send the URL of the new page to you, so you can comment the page.

Discussion: Systems like Connotea (http://www.connotea.org) or WIKINDX (http://wikindx.sourceforge.net/) support groups of students in collecting literature summaries.

This pattern is closely related to the READ, READ, READ pattern [3] which emphasizes the process of creating a literature summary.

Related Patterns:

- 2.6 DIARY: ADVISED LITERATURE RESEARCH as well as DIARY can help to document the student's progress.

2.8 TEST THE WATERS

Context: The student is working on the thesis project and manages her/his own schedule.

Problem: Thesis projects, as most research-related endeavors, contain many uncertainties. Many different activities must be undertaken by the student, e.g., literature research, reading papers, conducting interviews, writing large portions of text, creating design prototypes or developing working software. **If the student has no experience with the tasks the thesis project requires, the estimation of a schedule becomes difficult for her/him and s/he may miss deadlines.** Furthermore, if you have a wrong impression of the student's abilities, you are in danger of expecting too much, which can result in a worse grade than necessary.

Forces:

- You have your own idea of how much time is needed for a specific subtask, but you do not know how long an average student needs.
- You have an idea of the amount of time needed by the average student, but do not know how much time the student involved in this particular thesis project will need.
- The final deadline of the thesis is fixed for formal reasons.
- Students often underestimate the time needed, especially for writing the final text of the thesis, feeling too comfortable while having still some months to go.

Solution: Let the student perform important activities on a small scale and make her/him measure the time that was actually needed to complete the task. Arrange to discuss these findings with the student, so you get an impression of the student's abilities. With such data you can learn about the student's time needs for this project, and you can tell other students how much time their predecessors needed.

If you are uncertain about the student's writing abilities, make her/him write part of a chapter and let her/him tell you the time s/he needed. If programming plays an important role in the thesis project, you can give the student a small assignment and document the time s/he needed for completing it.

Technology Support: The student regularly sends you e-mails to inform you about the time needed for different tasks in the project. You evaluate this data and compare it with that of other students.

Discussion: In this pattern, we project the planning mechanisms of agile methodologies (e.g., [1, 4]) onto thesis projects. The essence of agile planning is that you can only estimate well based on first hand experience.

The name TEST THE WATERS is taken from a similar pattern by Manns and Rising [8].

Related Patterns:

- 2.5 STUDENT-MANAGED SCHEDULE: A change in the work pace should result in an updated schedule.

2.9 EXPRESSIVE STUDENT

Context: The student is working on the thesis.

Problem: The student doesn't have a good idea of how to describe what s/he is doing. S/he has problems to find the appropriate level of detail and does not have a clear picture of which aspects should be put into or left out of the written thesis.

Forces:

- Many things happen during a thesis project that are necessary for the process, but not for the final thesis.
- If a student is deeply immersed in her/his subject, s/he can lose her/his view of the big picture.
- Other people than the student and you have a different opinion of the subject of the thesis; this is especially relevant if these people are also responsible for the final grade.

Solution: Let the student express the subject of the thesis, both orally and in written form as often as possible and on different levels of granularity.

Have a mandatory *defense* of the thesis at different stages of the project: possibly after the Exposé has been written, before the student starts to implement a solution, and at the end of the project.

The defense should clearly state:

- the importance of the problem;
- the current state-of-the-art;
- approaches that the student wants to take; and
- the expected contribution and benefits.

Invite members of your research group as well as peer students to the presentation. Also ensure that students who are currently beginning to work on their thesis have a chance to attend a defense by a student who is at a later stage.

Make the student prepare an *elevator talk*: Tell a knowledgeable stranger in 30 seconds (about the time being together in an elevator) what the core ideas of the thesis are. This talk can be updated regularly during the project.

Make the student write and present an *incremental synopsis*, i.e. the core ideas of the thesis

- in one sentence;
- in one paragraph; and
- on one page.

The extended version of the one-page synopsis should be the exposé, if one was written upfront. Again, the incremental synopsis can be updated regularly throughout the project.

Make several students work together on their theses (in a *thesis project*, as described, for example, in [9]). Make them exchange ideas and let them help each other (pair programming, feedback on exposés, etc.). This can be very helpful in large research groups.

Technology Support: The incremental synopsis can be stored as a dedicated wiki page. The student should be able to reproduce the content of the synopsis in different computer-mediated communication settings, such as electronic mail (explaining the thesis in one paragraph when, e.g., inviting a secondary advisor) or text-based chat tools.

Discussion: It is important that the student defends the current status of the project, not you. The defense can take place in front of the whole research group or just with the student, the advisor and the professor.

If the discussion went well, the student will be convinced that the topic is worth working on (and not just the advisor providing the topic). Otherwise, the audience will provide useful hints for adapting the topic. The student gets trained in defending project proposals (important both in academia and in the industry).

The concept of a defense is, e.g., practiced at Fraunhofer IGD in Darmstadt by Peter Tandler, who proposed to include it in this pattern language. It is quite well-established at many US universities.

Related Patterns:

- 2.3 AGILE EXPOSÉ: The exposé can be used as an input for the incremental synopsis.
- INTROVERT – EXTROVERT [2] discusses the difficulties some people have with presenting themselves, their ideas, and their project to others. The pattern provides hints for introverted students, so that they will present their work more self-confidently.

3. Conclusion

This paper is intended as a first step towards making the interaction between students and supervisors more reliable and transparent. Initially thought as a paper that describes the interaction between supervisor and student at a distance teaching university, we discovered large commonalities with the ways such projects are run at traditional universities. We also observed that – although we did not attend the same universities at any point in time – there is an implicit agreement on how successful thesis projects should look like. The same applies to failed thesis projects that were not well supervised.

With this paper, we hope to initiate a broader discussion on good practices for supervising thesis projects. More high-level theses as well as less drop outs would justify our work.

Acknowledgements: Many people have helped us to write this paper. First of all, we would like to thank our numerous students who have suffered from our former way of advising thesis projects. Their pains made us look deeper into the problems and iteratively improve the interaction between supervisor and student. We also thank our recent students, since they made us more confident that we have found good patterns of interaction by now. Additional thanks are due to our colleagues who shared and discussed their style of supervision with us. We thank Peter Tandler for his initial comments and his view on thesis projects and especially for his input regarding the EXPRESSIVE STUDENT pattern.

We especially thank Andreas Rüping for shepherding this paper for EuroPLoP 2008.

4. References

[1] Beck, K. and Andres, C. *Extreme Programming Explained - Embrace Change (2nd ed.)*. Addison-Wesley, 2004.

[2] Bergin, J., Introvert - Extrovert. In *Proc. EuroPLoP '02*, UVK Konstanz, Irsee (Germany), 2002.

[3] Bergin, J.: Patterns for the Doctoral Student, http://pclc.pace.edu/~bergin/patterns/DoctoralPatterns.html, last updated: July 15, 2002, (last visited June 10, 2008).

[4] Cockburn, A. *Agile Software Development*. Addison-Wesley, Boston, 2002.

[5] Coplien, J.O. and Harrison, N.B. *Organizational Patterns of Agile Software Development*. Prentice Hall International, 2004.

[6] Deininger, M., Lichter, H., Ludewig, J. and Schneider, K. *Studien-Arbeiten (5. Aufl.)*. Vdf Zürich - B. G. Teubner, Stuttgart, 2005.

[7] Derntl, M., *Patterns for Person-Centered e-Learning*, Ph. D. thesis, Faculty of Computer Science, University of Vienna, Vienna, 2005.

[8] Manns, M.L. and Rising, L. *Fearless Change*. Pearson Education, Boston, MA, 2005.

[9] Olsson, B., Berndtsson, M., Lundell, B. and Hansson, J., Running Research-Oriented Final Year Projects for CS and IS Students. In *Proc. ACM SIGCSE*, Reno (Nevada), 2003, 79-83.

[10] Schümmer, T. and Lukosch, S. *Patterns for Computer-Mediated Interaction*. Wiley & Sons, 2007.

ACTIVATING STUDENTS IN INTRODUCTORY MATHEMATICS TUTORIALS (EuroPLOP 2008)

Pattern for introductory mathematics tutorials following a constructivist approach

Christine Bescherer and Christian Spannagel
University of Education Ludwigsburg

Wolfgang Müller
University of Education Weingarten

Abstract. *This paper describes a pedagogical pattern for mathematics tutorials with two different solutions depending on the underlying philosophy of learning or teaching. The aim of tutorials for introductory mathematics courses is for students to practice and apply what they have learned during the lecture. Adopting the traditional approach tutors show how to solve the given problems. Students observe the tutor solving problems on the chalkboard, copy the solution, and usually assume they are able to solve similar problems by themselves next time. Following a constructivist philosophy of learning we present a 'parallel' – different – solution to the same problem where learners do actively mathematics while learning how to solve mathematical problems.*

Preliminary remarks

This pattern describes design decisions which 'designers' of mathematics courses have to take. There is a long tradition of teaching mathematics to freshman at universities. In the pedagogical discussion after the TIMSS and PISA studies researchers in mathematics education recommend teaching and learning scenarios in schools where learners actively **do** mathematics, solve complex problems, reason and communicate mathematically, and make connections between different fields and topics. This concept differs widely from traditional tutorials at the universities.

In this paper we state the context and problem and then, present two different solutions based on different philosophies of teaching and learning. One solution follows the traditional approach, the other is based on the constructivist philosophy of learners actively building their own knowledge.

In presenting the patterns, we merge the traditional pattern format and a formalism developed by Wippermann (2008) especially for e-learning scenarios. In our experience, this formalism communicates the main ideas of an educational setting better than patterns designed for technological areas.

Context

Tutorials for introductory mathematics usually support mathematics lectures. In Germany there are often several hundred students in 'Introductory Math' courses at universities. The presentation of the lecture is usually given by a professor or lecturer. The tutorials differ widely in the number of attending students but usually there are smaller numbers of students (20 to 40 students in each group).

Proceedings of the 13th European Conference on Pattern Languages of Programs (EuroPLoP 2008), edited by Till Schümmer and Allan Kelly, ISSN 1613-0073 <issn-1613-0073.html>.

Tutorials normally don't introduce new topics but concern themselves with practicing the previous lectures' contents by solving mathematical problems given in worksheets.

The discussed setting in this paper is an introductory mathematics course for students who want to become math teachers for primary and lower secondary schools. This is one of the main reasons we follow a non-traditional teaching philosophy. As future school teachers these students have to gain enough experience learning in constructivist learning scenarios to be able to teach in these scenarios as well.

The pattern for constructivist mathematics learning can be easily adapted to other school forms (i.e. high school) or even subjects with similar processes and competencies like theoretical physics or computer science.

Problem / Challenge / Motivation[1]

Freshmen usually have to get used to the difference between mathematics at school and at universities. At universities, performing mathematics means much more than just solving a predefined set of mathematical problems in a given thematic context (e.g., arithmetic or geometry). It means applying solution strategies and problem solving heuristics such as finding examples and counter-examples, making conjectures, and drawing graphs. In addition, performing 'real' mathematics often means solving problems with no pre-defined single solution. Therefore, students have to decide which information is relevant for the solution, how to process this information, and how to present the results. In addition, they have to choose appropriate tools like spreadsheet calculators or dynamic geometry systems.

Especially future teachers should experience this kind of performing mathematics very early in their studies. They do not have to focus on the product (the solution of a problem), but on the mathematical processes necessary for solving the problem. The latter is the 'real' objective of learning mathematics. This change in the view on mathematics education is clearly stated, e.g., in the NCTM 'Principles and Standards for School Mathematics' (2000) where the process standards *problem solving*, *reasoning and proof*, *communication*, *connections*, and *representation* are considered as important as the content standards *number and operation*, *algebra*, *geometry*, *measurement, data analysis and probability*.

How do beginners learn these strategies efficiently?

Forces

Here is the point where the pattern follows two different paths to reach answers to the above stated challenge. Depending on the fundamental philosophy of learning and teaching two different solutions arise. The descriptions are given parallel in the following table to simplify the comparison.

The learning/teaching philosophies are deeply intertwined with theoretical pedagogical approaches which will also be described in the table (s. 'Rationale').

[1] In a pedagogical context the word 'problem' is not really adequate here. Students learning geometry is not a problem which can be solved once and for all – it's more a 'challenge'.

Traditional teaching philosophy	Learner activating teaching philosophy
"Mathematics is learned by being exposed to definitions, theorems, proofs, techniques and examples, through which one is exposed to formalization, proof, modeling, techniques etc. The teacher's job is to lay out the material clearly and logically. Students must rehearse many examples in order to develop facility and through facility, gain understanding of the concepts, the techniques and why the techniques work. Hard work is valued, work consisting largely of working through notes and problems to try to understand them." (Holton, 2001, 73) This is also the kind of learning most of the professors at German universities experienced when learning mathematics themselves.	"Mathematics is learned by reconstructing for oneself what others have thought and tried to expound clearly and logically. Reconstruction is carried out through constructing special and illustrative cases, trying to see generality through the particular, guided by theorems and other exposition. Exposition and practice on exercises is useful, but only as means to reconstruction. Facility and understanding grow together, as each contributes to the other and neither necessarily precedes the other." (Holton, 2001, 73) The basic idea of this concept is to motivate students actively doing mathematics. The students work on their own choice of problems, organized in small groups and aided by tutors. There are several forms of feedback during the tutorials as well as virtually in learning platforms.

Solution

Students **work on a set of given problems** and follow **the demonstrations of solved problems** and proofs of theorems during the tutorial. ***Planning and preparation:*** The **person responsible (lecturer or advanced tutor)** creates problem sheets containing a number of problems all of which are supposed to be solved by all students.	Students **pick from 5-6 weekly problem suggestions** and work in groups during the tutorial on the chosen problems **guided by the tutor** who doesn't give the correct solution away. ***Planning and preparation:*** The **person responsible (lecturer or advanced tutor)** creates problem sheets containing problems with the same mathematical topics but in different contexts for the students to choose from. The wording focuses on the solution process rather than the correct solution.

A typical problem:

Show that for any x, y, z $\in \mathbb{R}$, | x - z| ≤ | x - y| + | y - z|.

An expected typical solution to the problem:

$|x - z| = |x - y + y - z| \leq |x - y| + |y - z|$ *(triangle inequality)*[2]

The problems are related closely to the content of the lecture. A sample solution also has to be created. The problem sheets are delivered to students one week or more before the tutorial session. Tutors get the problem sheets at the same time plus a sample solution to prepare for the demonstrations.

A typical problem:

Make conjectures of several unit fractions concerning their decimal representation. What kind of decimal do you get?

If it is not a terminating decimal: How long are the periods and the delays of the periods? Make conjectures on the base of your data. Which properties determinate the kind of decimal? Which properties determinate the length of the period and the delay? Test your hypotheses with other unit fractions.

Hints / techniques:

- *You can use the Excel spreadsheets available in Moodle.*
- *Which of the unit fractions are good indicators for your conjectures?*

Expected activities:

The students are expected to try to understand the properties of decimal numbers using spreadsheets and to find the significance of denominators only containing powers of 2 and 5 compared to denominators without 2 and 5 or 'mixed' ones.[3]

The problem suggestions have to cover enough of the mathematical content so that students can not evade basic concepts such as reasoning and proof, finding examples, or special techniques like using group tables or important mathematical content. Some useful hints and references which don't give away too much have to be added.

Tutors need to be provided with ideas, hints, and strategies for exemplary solutions and problem solving. Also, they have to be

[2] Explanations for non-mathematicians: $|x - z| = |x - y + y - z|$ b*ecause* $-y + y = 0$ *for any* $y \in \mathbb{R}$ *and the triangle inequality is* $|x + y| \leq |x| + |y|$ *which is true for all* $x, y \in \mathbb{R}$

[3] All the fractions with denominators which consists only of powers of 2 and 5 (e.g. 1/40 = 1/(5*2³)=0.025) are terminating. All the fractions with denominators which consist of numbers without the factors of 2 and/or 5 are periodic (1/33=1/(3*11)=0.030303…) and the rest are delayed periodic (1/12=1/(3*2²)=0.083333…).

Tutors read through the given sample solution to be sure that they understand everything. **Students** work on the problems before the tutorial session. This is not monitored or supported by tutors.	briefed about aspects of insufficient solutions and problematic problem-solving strategies. Usually even tutors don't get the 'correct' solution from the lecturer. In fact, they have to think through the problems on their own and actively 'do mathematics'.
During the tutorials: The **tutors** present the problem solutions on the chalkboard. **Students** watch the demonstration and compare the solutions with their own (if they have created one). Alternatively, a student may be asked to present her or his own solution. Students may ask questions and discuss different solutions.	***During the tutorials:*** **Groups of students** start work on the problems during the tutorial session. They choose the problems they want to work on, discuss ideas, find examples, and verify or disprove statements on the worksheets and of others. They can use every tool they want to support the problem solving process - laptops, calculators or whatever. If necessary they ask for help and explain their problems. Finally they have to decide whether something is a solution or not. If they can't find a solution, they can also ask for help on their problem in discussion forums in the online learning platform. Then other students, tutors, or the lecturer may assist. The **tutors** give feedback, ask helpful questions and confirm that a solution is ok but don't give the correct solution. They give hints or mirror back some ideas and questions brought already up by the students. Very often, they just 'sit around' and do nothing.
After the tutorials: The sample solutions are given to the students. **Students** try to transfer the solutions to similar problems (reproduction). They practice under their own steam.	***After the tutorials:*** **All the participants** join in the online discussion. **Students** finish not yet solved problems and continue working on them based on their lecture notes, trying to connect them to problems they worked on during the tutorials.

Rationale

Theoretical Background	**Theoretical Background**
Basically this kind of teaching is experts demonstrating learners how mathematics "goes". The social cognitive theory (Bandura, 1989) states that people may learn by observing others doing something. The process of demonstrating a specific behavior is called *modeling* (Bandura, 2001). An example for this is a person in a restaurant struggling to eat a lobster. Observing other guests, this person is able to perform the task by imitating other guests' procedures. Vicarious experience is one source of self-efficacy expectations (Bandura, 1998). People who see others performing well may think that they could also master the task (Schunk, 1999; Margolis, 2005). In the cognitive load theory (Chandler & Sweller, 1991; Sweller, van Merriënboer, & Paas, 1998) instructional guidelines are developed which help to avoid irrelevant cognitive load during problem solving (Anderson, 1995). In many European universities mathematics lectures and tutorials are designed following this idea. Students follow an expert presenting the 'published' mathematics[4] in a very condensed way, avoiding wrong turns (Holton, 2001).	The basic idea of this concept is to motivate students to actively do mathematics. It is based on constructivist teaching philosophy insofar as by actively dealing with the mathematical problems themselves students gain experiences and insights and adjust their ideas and mathematical concepts which is all part of building new knowledge. Constructivist teachers[5] believe strongly in the idea that students construct knowledge for themselves and they will not truly learn something until they spend a good deal of time asking questions and actively thinking about the topic. The job as a teacher is to provide context, motivation and guidance. According to the cognitive apprenticeship model (Collins, Brown, & Newman, 1989) the latter has to fade with the learner's increasing expertise. Motivation is a crucial factor for learning. Intrinsically motivated students show higher levels of cognitive engagement in tasks than students who are more extrinsically motivated (Pintrich & Schrauben, 1992). Ryan and Deci (2002) state that three factors promote intrinsic motivation: *perceived choice*, *perceived competence*, and *related-ness*. The perception of competence is also related to the construct of self-efficacy (Bandura, 1998). Mathematical self-efficacy is the belief of a person that she or he is able to solve a mathematical problem (Betz &

[4] 'Published' means here the way mathematics is displayed in books where e.g. the concise and elegant form of a proof is printed and not the easier to follow but longer approach which shows how the proof was found the first time.
[5] For some first ideas on constructivism in mathematics education see e.g. http://mathforum.org/library/ed_topics/constructivism/, last visited May 31st, 2009

	Hackett, 1983; Pajares & Miller, 1995). One major source of efficacy expectations is performance accomplishments. Repeated successes have a positive influence on self-efficacy. Only if students try to solve the problems on their own there is a chance to increase their self-efficacy based on performances.
Reflections This is a very efficient way – from the teacher's point of view – to teach large groups of students how to solve mathematical problems. If 'an expert' (tutor or good student) demonstrates his previously prepared solution all students have seen at least one correct way of solving that particular problem. This solution is often used in introductory mathematics courses at universities and since a lot of mathematics teachers went through this system they obviously were successful. But there are a lot of drawbacks in this solution. 1. Non preparation Often students come to the tutorial session without own solutions or without even having read the problem sheet. Therefore they just copy the solutions without any understanding. Students need a lot of time and effort just before the exams to 'catch up' with all the missing understanding and often there just isn't enough time. There are possibilities to deal with this non-preparation of students and force them to work on the problems before they get the solution presented: At the beginning of the session a list with all problems is passed around and students have to tick the problems they have prepared. The tutor then calls students according to the list to present the solution. Every student has to tick at least 50% (or more) of all	**Reflections** Students have to get used to this kind of tutorial. A lot of the students are not very confident that they can really recognize a correct or incorrect solution. They want always some authority to check their answers. For these students there are several support structures beside the weekly face-to-face tutorials and online discussions. For example, there is an 'open math room' three to four times during the week where tutors answer questions. Learning to be a good problem solver requires working on problems at the same time as reflecting on the problem solving processes. This is very easy in collaborative settings and by actively participating in the problem solving processes. Working in groups students automatically communicate, ask questions, represent mathematical ideas, etc. and therefore mathematical processes can be experienced, reflected and discussed. One issue is definitely the time needed by the tutors for preparation and feedback.

problems otherwise he or she is not allowed to take the exam. Also, each student has to present solutions a certain number of times during the semester.

Another approach is to state problems that should be done at home by the students. Written solutions will have to be handed to the tutor at the beginning of the session. These solutions are graded (or just a feedback is given) and also an average grade must be reached to take part in the exams.

2. Illusion of Understanding
Presenting solutions without eliciting deep processing often creates the 'illusion of understanding' (cf. Atkinson et al., 2000). Students assume they have understood the solution. Realization that they didn't often occurs during the final exam.
This problem can be mitigated if students *actively process* the demonstrations (cf. Mayer, 2004). Some guidelines for the design of worked examples have been developed to increase the student's cognitive activity, e.g., giving incomplete worked examples (*completion problems*; Sweller et al., 1998), emphasizing the structure of the solution (Catrambone & Holyoak, 1990), or prompting students to elicit self-explanations (Chi, de Leeuw, Chiu, & LaVancher, 1994).

3. Motivation
Predominantly, students are extrinsically motivated. They want to pass the final test. But intrinsic motivation is a crucial factor in learning: see 'Theoretical background' of the 'Learner activating teaching philosophy – solution'.

4. 'Modern' learning theories
See: 'Theoretical background' of the 'Learner activating teaching philosophy – solution'.

Examples

This kind of tutorials can still be found at many German universities.	This kind of tutorials were realized from October 2007 until July 2008 at the University of Education Ludwigsburg in the courses Introduction to Arithmetic for Secondary Teachers (Einführung in die Arithmetik für Lehramt Realschule) and Introduction to Geometry for Secondary Teachers (Einführung in die Geometrie für Lehramt Realschule). The whole setting of weekly lectures and tutorials were evaluated by different instruments: a questionnaire on mathematical self-efficacy and a questionnaire on learning motivation. The results are published in Bescherer and Spannagel (2008; in German). Since October 2008 these tutorials are developed further in the context of the research project SAiL-M (www.sail-m.de) funded by the German Federal Ministry of Education and Research.

Related Patterns

	PATTERNS FOR ACTIVE LEARNING by Eckstein, Bergin, Sharp (http://www.pedagogicalpatterns.org/current/activelearning.pdf) TECHNOLOGY ON DEMAND, HELP ON DEMAND, FEEDBACK ON DEMAND (all Bescherer & Spannagel, 2009) and HINT ON DEMAND (www.sail-m.de)

Summary

The above presented pattern gives two solutions to the same challenge (problem) based on two different teaching philosophies. Of course these solutions describe more or less the extreme variations of tutorials and all shades in between these are possible.

Acknowledgements

We would like to thank Marc Zimmermann for final readings.

Literature

Anderson, J. R. (1995). *Cognitive Psychology and its Implications*. New York: W. H. Freeman and Company.

Atkinson, R. K., Derry, S. J., Renkl, A., & Wortham, D. (2000). Learning from Examples: Instructional Principles from the Worked Examples Research. *Review of Educational Research 70*(2), 181-214.

Bandura, A. (1989). Human Agency in Social Cognitive Theory. *American Psychologist 44*(9), 1175-1184.

Bandura, A. (1998). *Self-efficacy. The Exercise of Control*. New York: W. H. Freeman and Company.

Bandura, A. (2001). Modeling. In W. E. Craighead & C. B. Nemeroff (eds.), *The Corsini Encyclopedia of Psychology and Behavioral Science* (pp. 967-968). New York: John Wiley & Sons.

Bescherer, C. & Spannagel, C. (2009). Design Patterns for the Use of Technology in Introductory Mathematics Tutorials. *Education and Information Technologies (in press),* Springer

Bescherer, C. & Spannagel, C. (2008). Aktivierendes Mathematik-Lernen zum Studienbeginn. *Tagungsband der GDM-Tagung 2008, Budapest,* online available http://www.mathematik.uni-dortmund.de/ieem/BzMU/BzMU2008/BzMU2008/ BzMU2008_BESCHERER_Christine%20&%20SPANNAGEL_Christian.pdf, last visited: May 31st, 2009.

Betz, N. E. & Hackett, G. (1983). The Relationship of Mathematics Self-Efficacy Expectations to the Selection of Science-Based College Majors. *Journal of Vocational Behavior 23*, 329-345.

Catrambone, R. & Holyoak, K. J. (1990). Learning subgoals and methods for solving probability problems. *Memory & Cognition 18*(6), 593–603.

Chandler, P. & Sweller, J. (1991). Cognitive Load Theory and the format of instruction. *Cognition and Instruction 8*, 293–332.

Chi, M. T. H., de Leeuw, N., Chiu, M.-H., & LaVancher, C. (1994). *Eliciting Self-Explanations Improves Understanding*. Cognitive Science 18, 439-477.

Collins, A., Brown, J.S., und Newman, S.E. (1989). Cognitive Apprenticeship: Teaching the crafts of reading, writing, and mathematics. In L. B. Resnick (Ed.), *Knowing, learning and instruction* (pp.453–494). Hillsdale: Lawrence Erlbaum Associates.

Holton, D. (2001). The teaching and learning of mathematics at university level. *New ICMI study series*, vol. 7. Dordrecht: Kluwer.

Pajares, F. & Miller, M. D. (1995). Mathematics Self-Efficacy and Mathematics Performances: The Need for Specifity in Assessment. *Journal of Counseling Psychology 42*(2), 190-198.

Pintrich, P. R. & Schrauben, B. (1992). Students' Motivational Beliefs and Their Cognitive Engagement in Classroom Academic Tasks. In D. H. Schunk & J. L. Meece (eds.), Student Perceptions in the Classroom (pp. 149-183). Hillsdale: Lawrence Erlbaum.

Margolis, H. (2005). Increasing struggling learner's self-efficacy: what tutors can do and say. *Mentoring and Tutoring 13*(2), 221-238.

Mayer, R. E. (2004). Should there be a three strikes rule against pure discovery learning? The case for guided methods of instruction. *American Psychologist 59*(1), 14–19.

NCTM - National Council of Teachers of Mathematics (2000). *Principles and Standards for School Mathematics*. Reston, Virginia, USA.

Ryan, R. M. & Deci, E. L. (2002). Overview of Self-Determination Theory: An Organismic Dialectical Perspective. In E. L. Deci & R. M. Ryan (eds.), *Handbook of Self-Determination Research* (pp. 3-33). Rochester, NY: The University of Rochester Press.

Schunk, D. H. (1999). Social-self interaction and achievement behavior. *Educational Psychologist 34*(4), 219-227.

Sweller, J., van Merriënboer, J. J. G., & Paas, F. G. W. C. (1998). Cognitive architecture and instructional design. *Educational Psychology Review, 10*(3), 251–296.

Wippermann, Sven (2008): Didaktische Design Patterns zur Dokumentation und Systematisierung didaktischen Wissens und als Grundlage einer Community of Practice. Saarbrücken: Vdm Verlag Dr. Müller.

Didactic Design Pattern „Highlights"

a pattern for peer-review.

Sven Wippermann
University of Education Ludwigsburg, Germany
Reuteallee 46
D-71636 Ludwigsburg
+49 7141 140 744
wippermann@ph-ludwigsburg.de

Abstract: The core intention of the pattern is to enrich the learner's perspectives by giving and receiving feedback through peer-review.

1. Introduction

The Design Pattern „Highlight" has been developed and used within a the master study course of Educational Leadership at the University of Education Ludwigsburg, Germany. The course is embedded into a blended learning architecture. The Pattern focuses on the E-Teaching aspects of the learning scenario and aims at capturing the *didactic* knowledge on how to use this method within an E-Teaching setting.

Feedback is often provided by the lecturer without referring to other students' works. Further more students are not used to give feedback on other students' results. This pattern captures a best practice on using a specific, didactic driven method within a learning environment and is therefore particularly useful for the following audience:

- *Teachers and lecturers* who want their students to gain a different perspective on a solution/topic.
- *Teachers and lecturers* who want to implement a formative assessment of their students.

Depending on the discipline lecturers are more or less used to enrich their teaching with digital media. In order to reach a broad audience of lecturers of all kinds of disciplines this pattern contains two parts each with a specific focus and level of abstraction:

On the one hand it focuses on a very technical and abstract perspective following the common pattern structure to submit the core intension (part 1).

On the other hand part 2 emphasizes a pedagogic view upon this topic to submit information which is needed for planning and using learning scenarios (Siebert 2006).

Special thanks to my shepherd Michael who encouraged me to merge both structures and gave brilliant feedback for improving this pattern. We had good discussions opening up new perspectives on this pattern. Thanks a lot!

2. The Pattern Language

This pattern is part of a pattern language which will include the following related patterns:

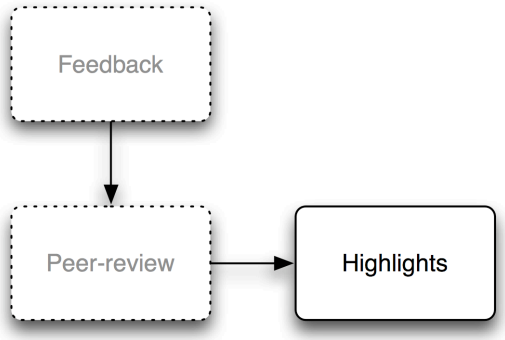


Figure 1: pattern language

3. Didactic Design Pattern „Highlights" (part 1)

Context

This pattern is effectively implemented within the master study course of Educational Leadership at the University of Education in Ludwigsburg, Germany. Within the module "personal management" this pattern provides a blended learning environment supporting the continuation of learning from the first face-to-face meeting. The students work on assigned tasks with the aim of exploring different points-of-view and a deeper level of inquiry into the subject matter. Following this broad and deep study, individuals create solutions and share them with their peers for reflection and comment. This interactive method fosters new views of the topic by exposure to a variety of understandings.

Problem

How can students gain different perspectives on solutions/tasks by providing feedback to one another?

Forces

Feedback plays an important role in regard to evaluating students' work, because it contains both positive elements and aspects that need to be improved in order to support the students' personal or academic development. Feedback is often provided by the teacher without referring to other students' works. Annotating students' work and giving feedback also increases the workload for teachers immensely.

Creating solutions, reviewing others' work, and receiving peer feedback allows students to explore new ideas and to gain a deeper and broader understanding of a given topic. Peer review provides the opportunity to learn from other students‘ work. Using this method teachers‘ workload is also significantly reduced.

Solution

In order to achieve this, **each student annotates the work of another student and returns it to the author who then picks one highlight to forward to the lecturer at a defined time.** He/she collects all highlights and publishes them to a learning management system.

A highlight in this sense is a concise portion of a solution that offers new insights into the given topic to reviewer. Due to the fact that every reviewer has a specific knowledge and point of view on the topic each highlight is very individual.

The solution invokes the following core activities (referring activities are explained within the implementation section, see part 2):

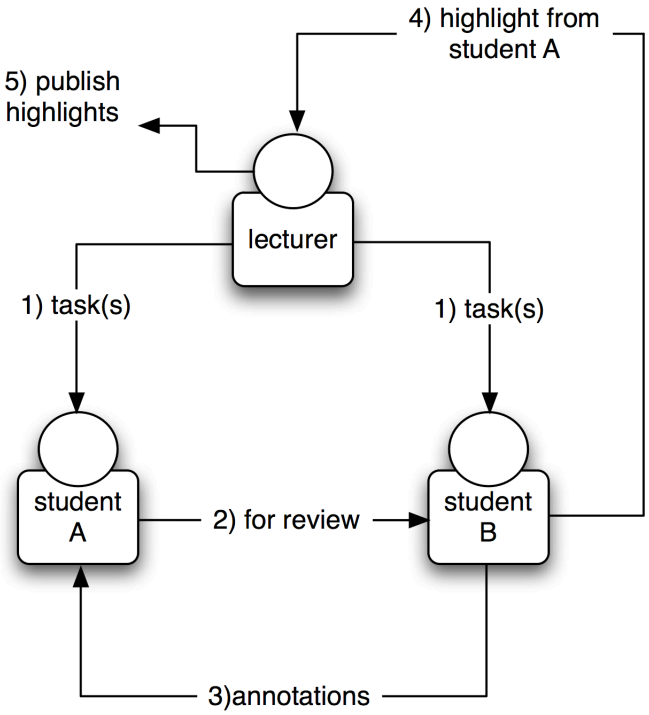


Figure 2: activities

Students work on the tasks they receive from the lecturer (1). The results of their work is forwarded to another student (student B) who acts as a reviewer (2). He/she annotates the work and returns it to the original author, student A (3). The reviewer also selects his/her highlight of the author‘s results and forwards it to the lecturer (4). The lecturer collects all highlights and publishes them (5).

The *teacher* acts as a coach, supporting the students in finding solutions to the given tasks. The *students* play the most active role within this phase by attending to the given tasks.

These activities are intended to make the students become aware of different approaches to the solutions, thus increasing their understanding and knowledge. These new understandings are brought about through the act of reviewing and annotating another students solution and viewing yet another peers' comments on their own solution.

Consequences

The benefits using this pattern are the following: Creating solutions, reviewing others' work, and receiving peer feedback allows students to explore new ideas and to gain a deeper and broader understanding of a given topic. Furthermore the students become aware of different perspectives by annotating their peer's work. Finally with peer reviewing being the primary mechanism of feedback, lecturers can devote more time to observing and fine-tuning the learning process.

The liabilities using this pattern include the following aspects: The pattern is centered on students creating their own solutions as well as reviewing and commenting upon other students' work -- which, in turn, is then evaluated by the lecturer (primarily through "highlights"). The addition step of peer review adds time to the process. Also students depend on one-another to complete tasks on-time. Thus, all students must respect the time-frame of each task in order to complete the pattern on schedule. Another liability can be found in the dependency on technical resources, especially the Learning Management System.

Discussion

Alternative usage may focus on two levels. In regard to an organizational level it is possible to hand the tasks to the students in a face-to-face environment with the advantage that questions can be answered directly, in plenum. In addition to that aspect assigning a single task to the students (instead of clustering many tasks), reduces the student workload for creating and annotating the solutions.
In regard to the activity level the highlights may also be sent to all students via email or in a face-to-face learning situation instead of publishing them within a learning management system. Finally the lecturer may skip adding to student annotations in the event that student annotations and highlights cover the target learning goals.

Known uses

This pattern is effectively implemented within the master study course of educational leadership at the University of Education in Ludwigsburg, Germany. Within the module "personal management" this pattern provides a blended learning environment supporting the continuation of learning from

the first face-to-face meeting. All tasks are clustered. From the complete set, students choose five tasks to complete utilizing the pattern.

This pattern also works within the pattern writing workshops. An author submits his/her pattern to peers who review it and give highlights to the author who explores new ideas of what he/she can keep or improve and also gains a deeper and broader understanding of how his work is interpreted.

In other educational contexts such as discussing a paper, students read through the text, highlight their key aspects and contribute their individual perspectives to the peers.

Related Patterns

Feedback loop (T. Schümmer)

References

Reich, K. (2002). Konstruktivistische Didaktik : Lehren und Lernen aus interaktionistischer Sicht. Neuwied [u.a.]: Luchterhand.

Siebert, H. (2006). Didaktisches Handeln in der Erwachsenenbildung : Didaktik aus konstruktivistischer Sicht (5. ed.). Augsburg: Ziel-Verlag.

Wippermann, S. (2008). Didaktische Design Patterns zur Dokumentation und Systematisierung didaktischen Wissens und als Grundlage einer Community of Practice. Saarbrücken: vdm.

4. Introduction to part 2

The second part of the pattern emphasizes a pedagogic view to submit information which is needed for arranging and using learning scenarios (Siebert 2006). It aims at supporting those lecturers who are not used to enrich their teaching with digital media and those who only have a weak affinity to such media usage by presenting essential pedagogical aspects.

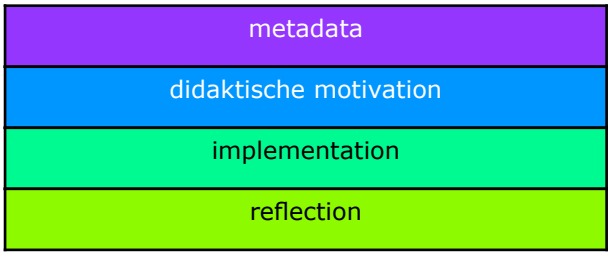
5. Structure of Didactic Design Patterns

Theoretical Background

These aspects refer the constructivistic didactic (Reich 2002). It contains the most holistic processes for arranging learning scenarios under a pedagogic perspective and is therefore essential for the Didactic Design Patterns (Wippermann 2008).

Meta-Pattern - new structure

The pedagogical elements are integrated into a new pattern structure (meta-pattern). The structure of each *Didactic Design Pattern* follows four main sections (Wippermann 2008):



A specific color represents each section on the right hand border of the pattern in order to help the reader navigation through it.

Each sections contains specific items to structure the knowledge within each pattern:

1. metadata

- name,
- date,
- status,
  - *draft version,*
  - *work in progress,*
  - *final version,*
- author,
- characteristics of E-Learning,
  - *communication vs. content centered,*
  - *synchronous vs. asynchronous,*
  - *independent on vs. dependent on special location.*

2. didactic motivation

- abstract,
- didactic motivation,
- hints for implementation,
  - *amount of learners,*
  - *social learning aspects,*
  - *state of learning,*
  - *time needed for implementation,*
  - *degree of competencies,*
  - *instruction vs. construction*

3. implementation

- didactic steps
  - planning and preparation,
  - information and instruction,
  - activities,
  - implementation,
  - evaluation,
- drama ,
  - *roles,*
  - learning activities,
- *tasks,*
- embedding,
  - *learning activities before pattern usage,*
  - *learning activities after pattern usage,*
- technical preconditions,
  - tools.

4. reflection

- problems,
- discussion,
  - *advantages,*
  - *disadvantages,*
  - *alternatives,*
- examples,
- references,
- related patterns.

Meta-Pattern - Characteristics at a glance

To provide an overview at the glance regarding didactic aspects each pattern starts with a visualization showing the characteristics of seven didactic items (see 2. didactic motivation).

The following table shows the variety of each item linked to a specific icon that allows a faster understanding (Wippermann 2008):

icon	name	characteristics
	e-learning	▪ synchronous vs. asynchronous, ▪ communication centered vs. content centered ▪ dependent on vs. independent on special location.
	amount of learners	▪ small, ▪ middle, ▪ huge.
	social aspects	▪ individual work, ▪ team work, ▪ group work, ▪ plenum.
	state of teaching/learning	▪ to start off, ▪ to work, ▪ to integrate, ▪ to evaluate.
	amount of time needed	▪ days, ▪ weeks, ▪ months.
	teacher`s competencies (in realizing the pattern)	▪ few, ▪ some, ▪ many.
	instruction vs. construction	▪ instruction, ▪ construction.

The characteristics of all items are gathered and visualized in order to provide selective knowledge of the pattern (see 6.).

Additional information about a pattern (version number, status, ratings) are also provided next to the characteristics stated above (see 6. and Wippermann 2008).

All of these information is essential for arranging learning scenarios and especially support lecturers who do not have a strong affinity to digital media in gaining an idea of the patterns‘ potential.

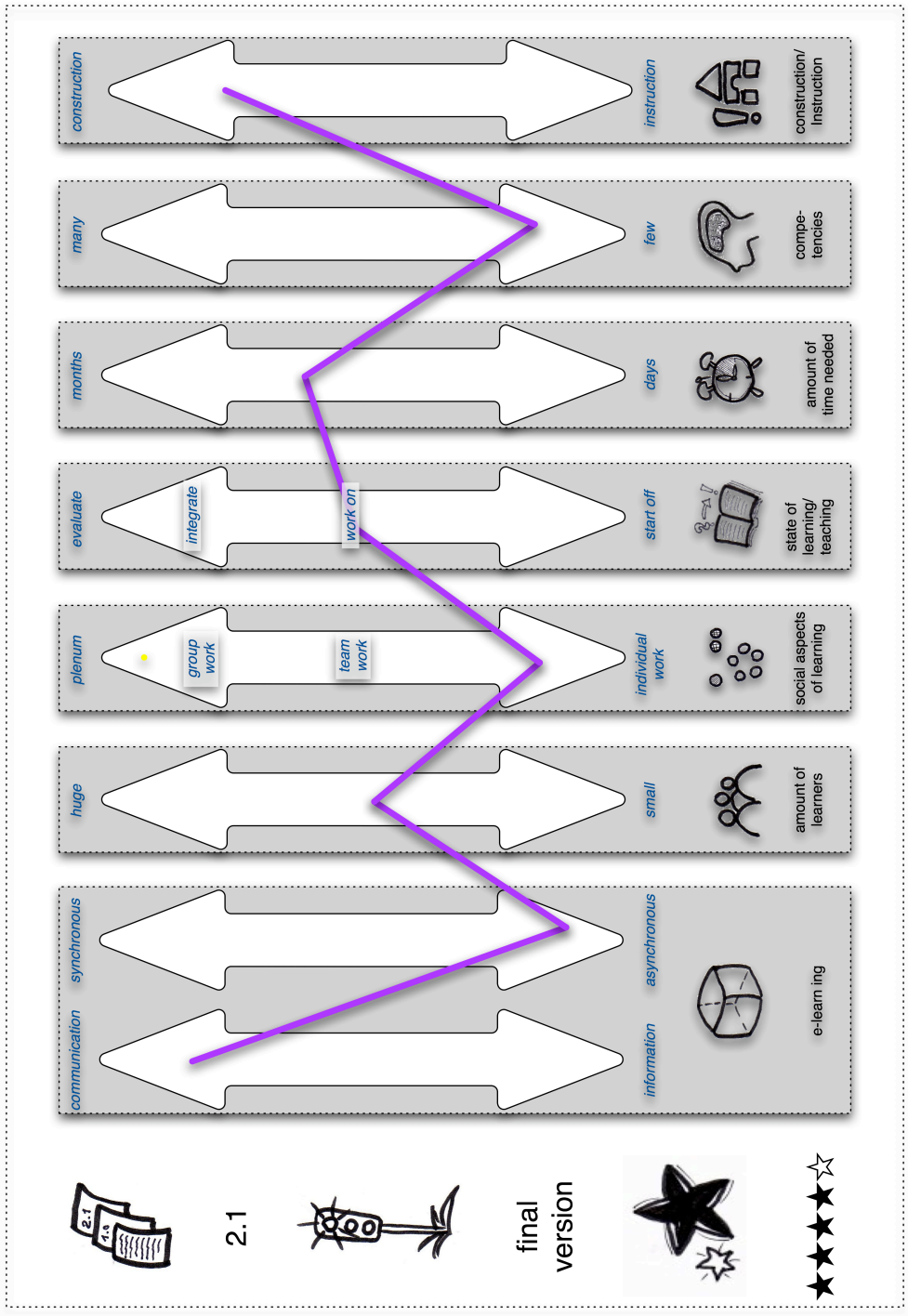
6. Didactic Design Pattern „Highlights „

a pattern for peer-review

- Sven Wippermann -

wippermann@ph-ludwigsburg.de

12.08.2004 - 12.02.2008



Didactic Motivation

The students work on assigned tasks with the aim of exploring different points-of-view and a deeper level of inquiry into the subject matter. Following this broad and deep study, individuals create solutions and share them with their peers for reflection and comment. This interactive method fosters new views of the topic by exposure to a variety of understandings.

Abstract

This pattern describes the handling of individual students' work results within a given time-frame: At a defined time each student sends his solution to another student who annotates the work and returns it to the author who then picks one highlight to forward to the lecturer. He/she collects all highlights and publishes them to a learning management system.

Implementation

1. **Planning and preparation:**

a. Conceptual design of tasks
The lecturer designs different tasks in regard to a specific learning topic and lso states his requests for the solutions (e.g. complexity of results etc.). He/she has to bear in mind that the student‘s results have to be composed on a computer in order to send them to another student quickly and easily -- and to be published within a learning management system.

b. Clustering of tasks
The lecturer unites the designed tasks (see above) to thematic clusters and defines the number of tasks within each cluster.

c. Definition of annotation mode
One process of assigning student reviewers is based on simple alphabetical sequence: Each student forwards his results to the student whose surname falls immediately after his/her own surname, alphabetically; the student at the end of the alphabetical list forwards the results to the first student on the list. The lecturer is responsible for the alphabetical student listing (which should include email addresses).
Lecturers must also define the specific type of feedback that students should focus their annotations upon (e.g. correctness of results, new view/insight on topic, etc.) and how the annotations should be formatted (e.g. below the result in a different color, etc.).

d. Definition of a highlight

The lecturer defines the meaning of the term "highlight". A useful working definition is "a concise portion of a solution that offers new insight into the given topic". Furthermore, the lecturer must also specify the number of highlights to be included with each solution. The number of highlights included with a solution should be a subset of the total comments provided (e.g., students may include as many comments as they wish, but they must select 3-5 comments to distinguish as highlights).

e. Definition of time-frame

The implementation of this pattern is based on a specified time-frame based upon the amount of work assigned to the students (see Tasks above.). The following deadlines have to be defined:

- date on which results have to be forwarded to next student (see c.) via email.
- date on which the foreign results have to be annotated and sent back to the author.
- date on which the chosen highlights have to be sent to the lecturer.
- date on which the highlights have to be published to the learning management system.

f. Preparation of learning management system

All highlights will be published to the learning management system. The lecturer is responsible for meeting the technical requirements and for ensuring that the system works properly (it might be necessary to set up a secure learning space for the course).

g. Initiating student work

A message providing instructions for the students should be composed. This message should include all necessary instructions as defined in the Planning and Preparation phase. (see Activity Phase).

During Planning and Preparation, only the lecturer plays an active role in specifying and designing the learning activities and setting-up the learning environment.

2. **Information and Instruction:**

The lecturer sends the Instructions and all necessary materials to the students via email.

The lecturer activities within this phase are: providing instructions to students regarding learning activities and responding to student questions.

The students occupy themselves with the given activities, posing questions as desired.

3. **Activities:**

a. Individual task activities
   The students work on the given tasks individually and compose their results via computer, respecting the specified deadlines.

b. Forwarding results to peers
   The results are properly annotated/commented and are then forwarded to the appropriate within the specified time (follow deadline).

c. Annotation of peer results
   Within the defined time-frame the results are annotated by the students following the annotation mode. Students should give special attention to potential highlights.

d. Return results with annotations to student
   The annotations are sent to the author within the specified time-frame so every student receives feedback on his results.

e. Send highlights to lecturer
   Following the schedule, the reviewing student choses the recommended number of highlights (among all annotations), and sends the highlights with the results and complete annotations to the lecturer.

f. Collecting highlights
   The lecturer collects the highlights and arranges them according to the task clusters.

Within this phase of implementation the lecturer acts as a coach, supporting the students in finding solutions to the given tasks.

The students play the most active role within this phase by attending to the given tasks. These activities are intended to bring students into awareness of different approaches to the solutions, thus increasing their understanding and knowledge. These new understandings are brought about through the act of reviewing and annotating another students solution and viewing yet another peers' comments on their own solution.

4. **Presentation:**

a. All highlights are summarized in one digital document.

b. This document is published within a specified area within the learning management system.

Here, the lecturer is active in the role of publishing the student‘s highlights.

5. **Evaluation:**

The lecturer should take the opportunity to provide additional commentary on the received highlights. These annotations may be included in the electronic document (see 4.), or published separately within the learning management system.

This pattern is incorporated in a specific learning context which consists of these sections:

1. Introduction section
   Tasks focusing on special topics that have to be introduced to the student.

2. Closing section
   This pattern ends with the publication of the student highlights. However, a discussion of the highlights will also support and extend the learning process.

The implementation of this pattern is linked with specific technical preconditions and may be supported by necessary tools.

1. Technical preconditions
   a. Email account,
   b. web browser,
   c. learning management system,
   d. valid account for learning management system,
   e. text editor or word processing application.
2. Tools
   a. Email account
      Free email accounts are available from yahoo.de, web.de, gmx.de or others.

b. Email client
   Most providers offer a web interface which can be used to sent mail. A free email client named thunderbird is available under this URL (http://www.mozilla.org/, retrieved 23.08.2007).

c. Web browser
   A web browser is installed on almost every computer. Free browsers are also available: firefox (http://www.mozilla.org/, retrieved 23.08.2007) opera (http://www.opera.com/products/, retrieved 23.08.2007), safari (http://apple.de/, retrieved 24.11.2007).

b. Learning management system
   A german version of the learning management system named moodle is available under: http://www.moodle.de/, retrieved 23.08.2007. A free trial account to BSCW (basic support for cooperative work / be smart - cooperate worldwide) may also be used as learning management system: http://public.bscw.de, retrieved 23.08.2007.

c. Valid account for learning management system
   All student must have a valid account in order to access the highlights.

d. Text editor or word procession application
   A free office suite is named Open Office is available here: http://de.openoffice.org/, retrieved 23.08.2007).

Reflection of the Didactic Design Pattern

1. **Potential problems**

   a. Technical problems

      1. See technical preconditions (it is advisable to save the documents in the rich text format).

      2. The tasks, results and annotation must be composed on a computer.

   b. Vague instructions

      1. The tasks must be written in clear, concise prose so that the students readily understand what to do. This is very important because --in contrast to face-to-face learning-- the lecturer has no opportunity to react directly to student‘s questions e.g. interpreting gestures, receiving and providing instantaneous clarification, etc.
         Questions about the structure of the pattern should be answered and forwarded to all students (such as in a Frequently Asked Questions document published to the Learning Management System).

2. The annotation mode must be specific and concise to facilitate forwarding results.

c. Management of deadlines

2. This pattern consists of different sections with specified time-frames to be defined, communicated and followed.

2. Reasons for not implementing this pattern

a. Insufficient technical resources.

b. Insufficient time to allow for proper implementation.

3. Advantages of the pattern

a. Creating solutions, reviewing others' work, and receiving peer feedback allows students to explore new ideas and to gain a deeper and broader understanding of a given topic.

b. The students become aware of different perspectives by annotating their peer's work.

c. With peer reviewing being the primary mechanism of feedback, lecturers can devote more time to observing and fine-tuning the learning process..

4. Disadvantages of the pattern

a. The pattern is centered on students creating their own solutions as well as reviewing and commenting upon other students' work --which, in turn, is then evaluated by the lecturer (primarily through "highlights"). The addition step of peer review adds time to the process.

b. Students depend on one-another to complete tasks on-time. Thus, all students must respect the time-frame of each task in order to complete the pattern on schedule.

c. Dependency on technical resources, especially the Learning Management System.

5. Alternatives

a. Organization

1. It is possible to hand the tasks to the students in a face-to-face environment with the advantage that questions can be answered directly, in plenum.

2. Assigning a single task to the students (instead of clustering many tasks), reduces the student workload for creating and annotating the solutions.

b. Activities

1. The highlights may also be sent to all students via email or in a face-to-face learning situation instead of publishing them within a learning management system.
2. The lecturer may skip adding to student annotations in the event that student annotations and highlights cover the target learning goals.

The following statement clarifies the implementation of the pattern:

This pattern is effectively implemented within the master study course of educational leadership at the University of Education in Ludwigsburg, Germany. Within the module "personal management" this pattern provides a blended learning environment supporting the continuation of learning from the first face-to-face meeting. All tasks are clustered. From the complete set, students choose five tasks to complete utilizing the pattern.

References:

No specific references.

Online glossary: http://www.e-teaching.org/glossar, retrieved 30.08.2007

The Didactic Design Patterns create a network and are related to each other

Related Didactic Design Patterns

- Use of Themenweb,
- virtual collaboration,
- virtual mood barometer.

Related Support Patterns

- StudienMail,
- Introduction Page of Learning Management System.

Guess my X and other techno-pedagogical patterns

Toward a language of patterns for teaching and learning mathematics

Yishay Mor (yishaym@gmail.com), London Knowledge Lab

Abstract

Most people see learning mathematics as a demanding, even threatening, endeavour. Consequently, creating technology-enhanced environments and activities for learning mathematics is a challenging domain. It requires a synergism of several dimensions of design knowledge: usability, software design, pedagogical design and subject matter. This paper presents a set of patterns derived from a study on designing collaborative learning activities in mathematics for children aged 10-14, and a set of tools to support them.

Introduction

This paper considers the question of designing educational activities for learning mathematics and the technologically-enhanced environments to support them. It is grounded in a techno-pedagogical approach with prioritises the function of educational technologies over their structure. This function is determined as much by their mode of use as it is by their static features; hence I see a discussion of tools as meaningful only with respect to the activities in which they are employed. My perspective is in line with the "instrumental genesis" approach to the design and analysis of educational technology (Ruthven 2008; Trouche, 2005; Folcher, 2003; Rabardel and Bourmaud, 2003; Trouche, 2003). As Rabardel and Bourmaud phrase it, "design continues in usage" (Rabardel and Bourmaud, 2003, p 666). In view of this position, the patterns presented in this paper do not

---

Proceedings of the 13th European Conference on Pattern Languages of Programs (EuroPLoP 2008), edited by Till Schümmer and Allan Kelly, ISSN 1613-0073 <issn-1613-0073.html>.

make a distinction between tool and activity. These patterns can be implemented by developing new software components, by configuring and orchestrating existing components or even – in some cases – with analogue technologies such as paper and pencil.

Target

This paper addresses designers of educational activities and the technology to support such activities. It takes an inclusive approach, following Herbert Simon's position that – "Everyone designs who devises courses of action aimed at changing existing situations into desired ones" (Simon, 1969, p 129). The focus of this paper is on designing opportunities for learning mathematics. Thus, the designers in mind are educational practitioners who are considering ways of using technology to enhance their teaching, as well as technology designers who wish to support innovative educational practices,. A third potential audience are researchers experimenting with new practices or technologies. Likewise, the notion of technological design extends beyond the production of new tools from scratch, and includes the appropriation and adaptation of existing tools to enable particular practices.

Therefore,

Educators could use these patterns in designing educational activities, using off-the self tools.

Educators and learners could use these patterns as a language for participatory learning design, where the learners are included as equal partners in the educational process.

Educators and developers could use these patterns as a language for discussing the design and development of new educational technologies.

Developers could use these patterns as an entry point into contemporary learning theories, by embedding these in the familiar form of a pattern language.

Historical background

The patterns below are derived from The *WebLabs* Project (www.weblabs.eu.com), which has been described in detail elsewhere (Mor et al, 2006; Simpson et al, 2006). The project aimed at exploring

new ways of constructing and expressing mathematical and scientific knowledge in communities of young learners. The WebLabs project involved several hundred students, aged ten to fourteen, across sixteen schools and clubs in six European countries. Our approach brought together two traditions: *constructionist learning* as described by Papert & Harel (1991) and collaborative *knowledge-building* in the spirit of Scardamalia & Bereiter (1994). The former was largely supported by the programming language ToonTalk (Kahn, 1996; 2004; Morgado and Kahn, in press) (www.toontalk.com), whereas for the latter we have designed and built a web-based collaboration system called *WebReports* (Mor, Tholander & Holmberg, 2006). The central design intention of our approach is that students should simultaneously *build* and *share* models of their emerging mathematical knowledge.

ToonTalk is an animated language and a programming environment designed to be accessible by children of a wide range of ages, without compromising computational and expressive power. Following a video game metaphor, the programmer is represented by an avatar that acts in a virtual world. Through this avatar the programmer can operate on objects in this world, or can train a robot to do so. Training a robot is the ToonTalk equivalent of programming. The programmer leads the robot through a sequence of actions, and the robot will then repeat these actions whenever presented with the right conditions. ToonTalk programmes are animated: the robot displays its actions as it executes them.

The *WebReports* system was set up to serve both as a personal memory aid and as a communication tool. *web report*is a document that is composed and displayed online, through which a learner can share experiences, questions and ideas derived from her activities. The uniqueness of our system is that it allows the author to share her ideas not just as text, but also graphics and animated ToonTalk models. This last point is crucial: rather than simply discussing what each other may be thinking, students can share what they have built, and rebuild each others' attempts to model any given task or object.

A main concern was the careful design of a set of activities, aiming to foster learning of specific mathematical topics, such as sequences, infinity and randomness. The choice and design of technologies was subordinate to this cause.

The Patterns

This paper assumes a pedagogical approach which combines construction, communication and collaboration. The patterns presented here are focused on this perspective. They were derived from a three-year European educational research project. These patterns are viewed as the first steps towards a language. Figure 1 offers a "fantasy map" of this language – a possible draft of the form it may eventually take. This map is by no means comprehensive. To cover the whole field of technology-enhanced mathematical education would be a project of immense scale. The patterns actually discussed in this paper are highlighted in this map. The lines in the map show the links between patterns to other patterns they extend or use.

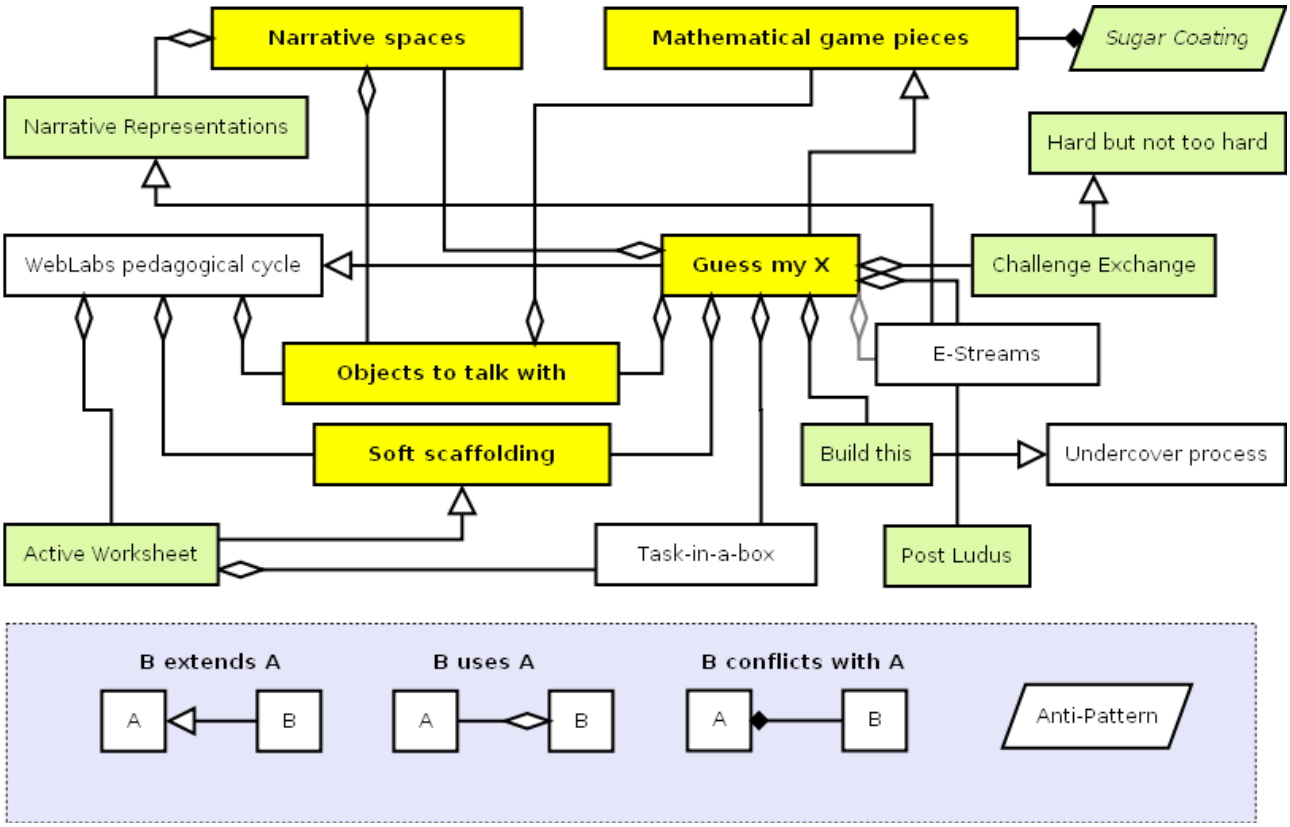


Figure 1: A "fantasy map" of the emerging pattern language. Dark highlighted patterns are presented in this paper. Light highlighted ones are mentioned. Italics denote anti-patterns.

The most elaborate of these is GUESS MY X (GmX), which was elicited from the guess my robot activity (Mor et al, 2006) and other activities cast in its mould by my colleagues Michelle Cerulli and Gordon Simpson. Other patterns were derived from this one as higher-order abstractions or as

components.

The five patterns described in detail are followed by “thumbnail” descriptions of other patterns mentioned in the text.

The structure of the design patterns aims to balance precision and simplicity. Visualising these patterns is often challenging, since they relate to intangible aspects of learning and teaching processes. I have tried to identify imagery which invokes a useful metaphor.

The problem descriptions utilize an innovative form of visual force-maps. These maps are based on Alexander’s force diagrams (1964), extended with iconic illustrations. While a formal methodological discussion of these force maps is called for, it is out of the scope of this paper. Is is hoped that these maps will prove of intuitive value and assist readers in understanding the problems under inspection.

1. Mathematical Game Pieces

Mathematical content is often injected artificially into games or other activities, as SUGAR-COATING. This has a dual effect of ruining the game and alienating the mathematics. By contrast, for many mathematicians, mathematics **is** the game.

The problem

How do you design (or choose) a game to convey mathematical ideas in an effective and motivating manner? How do you judge if a proposed game is an adequate tool for teaching particular mathematical concepts?

Forces

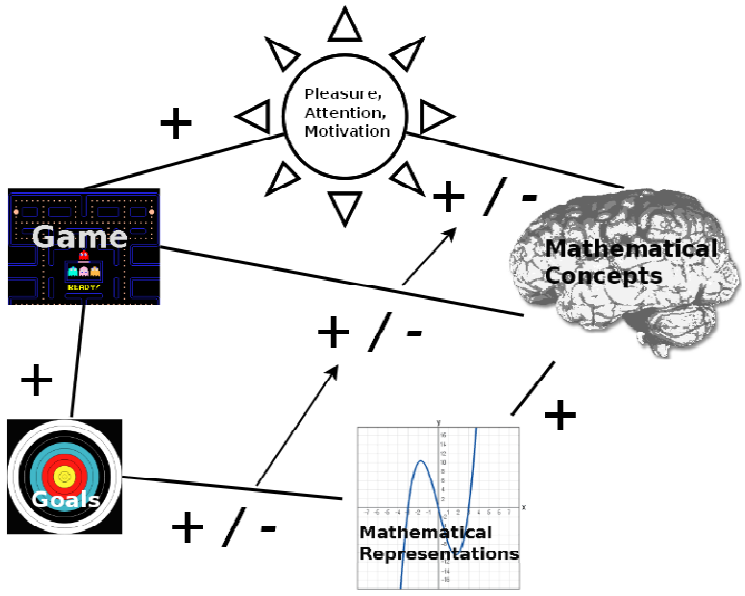


Figure 2: Force map for Mathematical Game Pieces pattern

- A game used in education has to provide a good game experience, or else it is "just another boring task".
- Learners need to engage with the mathematical content that the game aims to promote.
- The chosen representations need to be consistent both with the game metaphors and with the epistemic nature of the content domain.

Context

An educator wishing to use games as part of her teaching, either evaluating existing "educational" games, appropriating "entertainment" games, or designing and developing her own games.

A developer wishing to develop games for the educational market.

Solution

- Identify an element of the mathematical content you wish to address in this game.

- Find a visual, animated or tangible representation of this element which is consistent with the game metaphors.

- Design your game so that these objects have clear PURPOSE AND UTILITY as game elements in the gameplay structure.

The objects representing the mathematical content should have a meaningful intrinsic role in the game. Manipulating these objects can be part of the game rules or goals, or understanding them could be a necessary condition for success.

If the game includes or is followed by communication between participants, then the mathematical game pieces should become OBJECTS TO TALK WITH.

Examples

In Chancemaker (Pratt, 1988) users manipulate the odds of various chance devices, such as dice and roulette wheels. The game pieces are representations of probability (Figure 3).

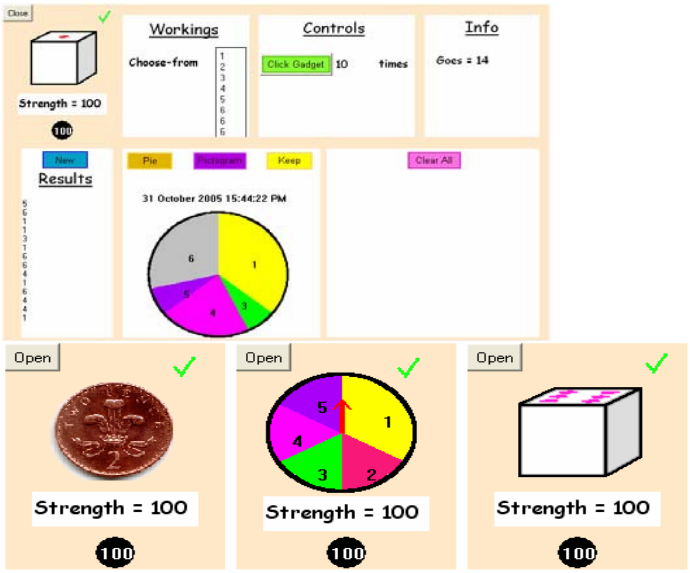


Figure 3: Chancemaker gadgets are probability game pieces

In Programming Building Blocks (http://www.thinklets.nl) the object of the game is to reconstruct a 3D shape from its 2D projections. The mathematical content is the focus of the game, and the objects used in the game are straightforward representations of that content (Figure 4).

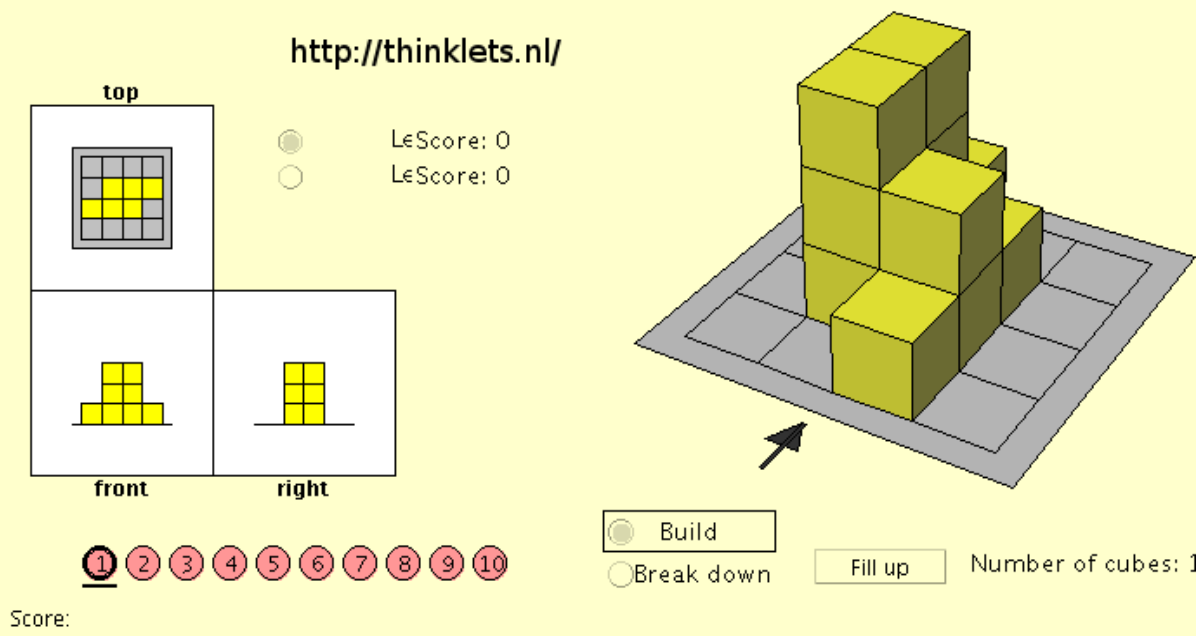


Figure 4: Building blocks constructions as 3D geometry game pieces

Related Patterns

Used by: OBJECTS TO TALK WITH.

Contradicts: *SUGAR-COATING* (anti-pattern)

Elaborated by: GUESS MY X

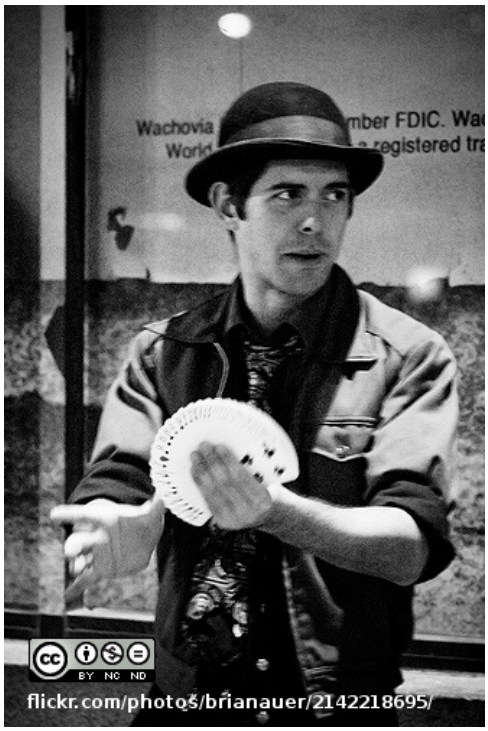
Notes

This is a very high level pattern which needs to be elaborated per specific game and content classes. For example, in a quest type game it might spawn different sub-patterns than in puzzle type games.

Likewise, factual and procedural knowledge might lead to different strategies than meta-cognitive skills. Nevertheless, it is useful as a guideline for evaluating design proposals. Its absence marks a game as a weak tool for learning.

2. Guess My X

Use a CHALLENGE EXCHANGE game of BUILD THIS puzzles to combine construction and conversation, promoting an understanding of process-object relationships and lead to meta-cognitive skills such as equivalence classes, proof and argumentation.



The problem

A teacher wants to design a game for learning concepts, methods and meta-cognitive skills in a particular mathematical domain. This game should use a combination of available technologies.

Many complex concepts require an understanding of the relationship between the structure of an object and the process which created it. Novices may master one or the other but find it challenging to associate the two. Constructing objects helps build intuitions, and discussing them espouses abstracting from intuitions and establishing socio-mathematical norms (Yackel & Cobb, 1995).

Learning mathematics is fundamentally learning to be a mathematician. It requires the learner to internalize a range of mathematical skills as regular habits: computation, analysis, conjecturing and hypothesis testing, argumentation and proof. For this to happen, the learner needs to be deeply engaged in meaningful mathematical inquiry, problem solving and discussion. Games provide a natural setting for the kind of "flow" needed, but how do we ensure that the focus of this flow is

mathematical activity and discourse?

Forces

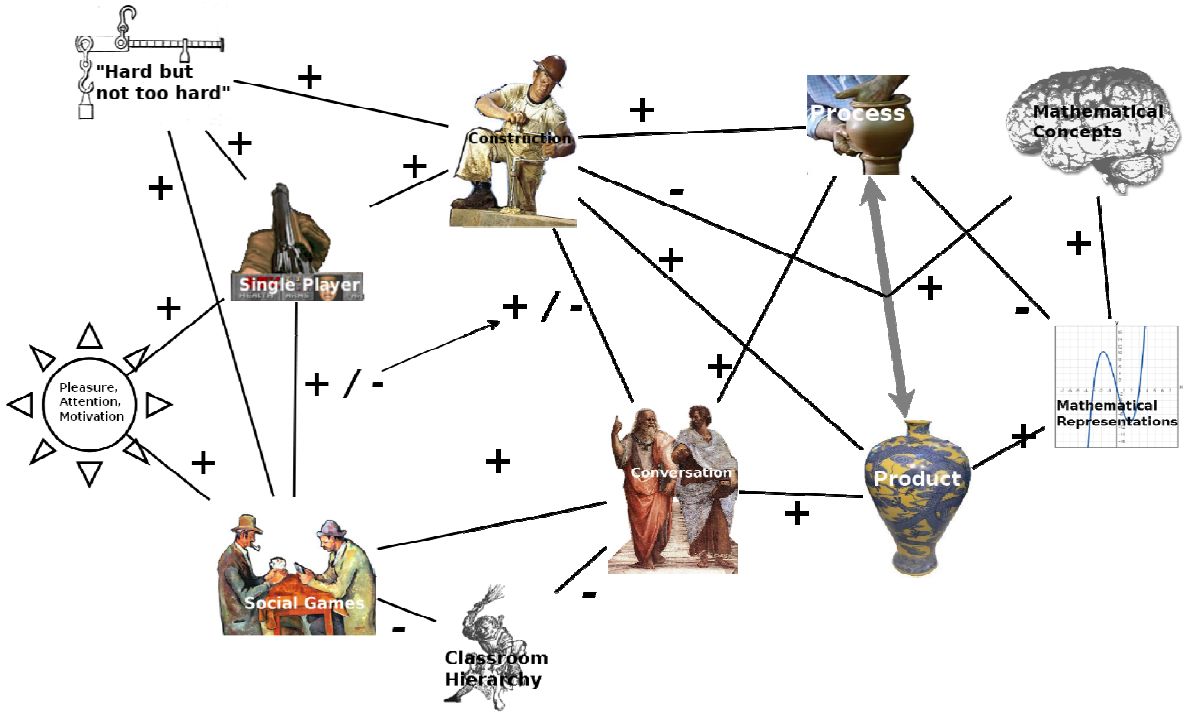


Figure 5: Force map for the Guess my X design pattern

- Many mathematical domains require learners to understand the relationship between mathematical objects and the process which generated them. This is a challenge for many learners.
- The teacher needs a non-invasive monitoring mechanism to assess students' performance.
- The communicational approach (Sfard, 2006) sees learning mathematics as acquiring a set of language rules and meta-rules. In order to achieve this, learners need to engage in meaningful and sustained discussion of mathematical topics.
- The classroom hierarchies and the perception of a teacher as more knowledgeable causes learners to be cautious and restrained in their mathematical discourse.

Context

Primarily, a classroom supported by a technological environment which provides a shared and protected web space (e.g. wiki or forum), common tools (programming environment, spreadsheets, etc.) and sufficient access time for all students.

The game relies on sustained interaction over a period of several sessions. It can be used as a short introduction to a topic, but the greater meta-cognitive benefits may be lost if not enough time is allowed for conversations to evolve.

Also works for several groups collaborating over a web-based medium.

Solution

Guess my X is a pattern of game structure, which can be adapted to a wide range of mathematical topics. It extends CHALLENGE EXCHANGE to encourage discussion and collaborative learning, and to break down classroom hierarchies. It uses BUILD THIS to engender reflection and discussion about the relationships between mathematical objects and the processes that produce them. The core of the pattern is described in Figure 6.

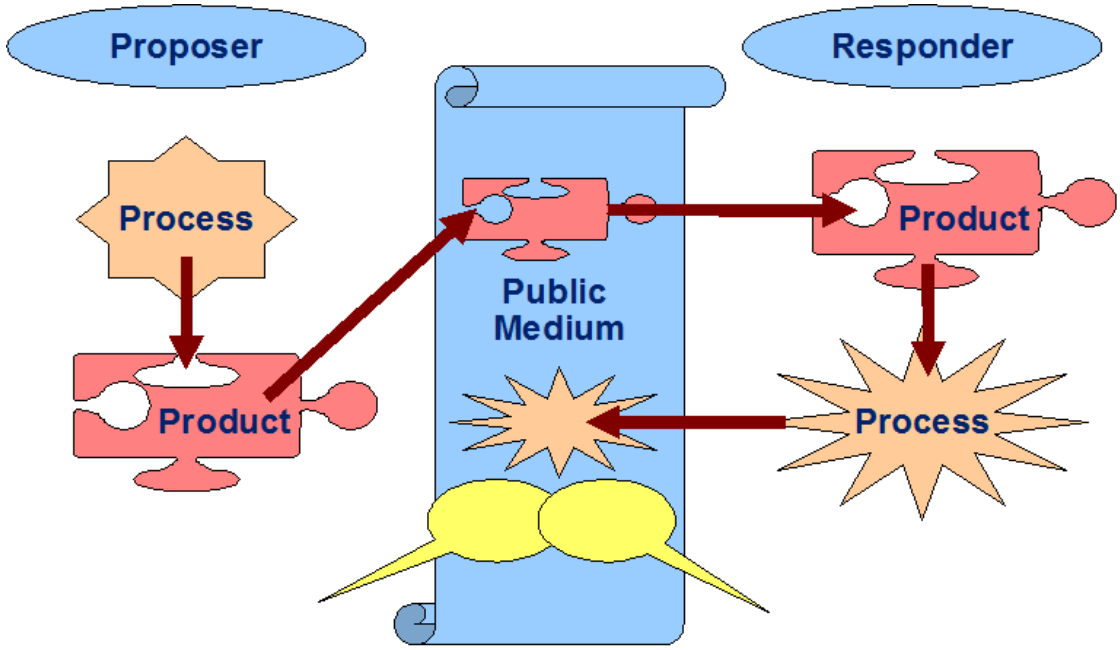


Figure 6: Schematic diagram of GmX

GmX involves players in two roles, *proposers* and *responders*, and a facilitator. An implementation of the game would specify a domain of mathematics and rules for constructing processes in that domain. A proposer sets a challenge, in the form of a mathematical object which she constructed. The explicit rules of the game define the nature of the process by which this object can be created, but not its details. A proposer would construct such a process, and capture its product. The proposer then saves the process model in a private space and publishes the product as a challenge. Responders then need to "reverse engineer" the process from the product. If they succeed, they publish their version as a response to the challenge. The proposer then needs to confirm the responder's solution of provide evidence for the contrary.

It is important to keep the mathematical content explicit from the start. The game is not a SUGAR-COATING to disguise the mathematics: it is a game with MATHEMATICAL GAME-PIECES. The rules of the game are intentionally left vague, in the sense that the evaluation function used to determine the responders' success is not fully specified. This requires students to negotiate what constitutes a correct answer, and in doing so collaboratively refine the underlying mathematical concepts. These negotiations can lead to discussions of issues such as proof, equivalence and formal descriptions. The quality and extent of these discussions depends on the scaffolding and provocations provided by the teacher, but a necessary condition for them to emerge is that the medium of the game provide

a NARRATIVE SPACE, where the MATHEMATICAL GAME-PIECES of the game can become OBJECTS TO TALK WITH.

Set-up phase

Before the game begins, the teacher needs to verify that the players have a minimal competence in analysing and constructing the mathematical objects to be used.

1. Teacher introduces the rules of the game and the game environment.
2. Teacher simulates one or two game rounds during a whole class discussion.
3. Students may need to initialize their game space on the chosen collaborative medium.

If the game uses separate media for construction and communication, consider using a TASK IN A BOX to streamline the transition between them.

Game session

The game sessions for the proposer and the responder are different, although the same player can play both parts in parallel.

1. Proposer initiates the game, by constructing and object according the game rules and publishing it. She then waits for responses.
2. Responder chooses an attractive challenge, and attempts to resolve it. If she believed she has succeeded, she responds to the challenge by posting the object she constructed and the method she used.
3. Proposer reviews the response, and confirms or rejects it. If the response is rejected, an argument needs to be provided.

Play session

Each play session involves a single iteration of the game. Students tend to prolong their interaction in the game, by providing secondary challenges, etc. Since the iterations are a-synchronous, there

may be a time gap of several days between turns.

The communication medium chosen for the game should afford NARRATIVE SPACES for the proposer and the responder. Although the rules of the game are limited to the exchange of mathematical objects, the ability to augment these with personal narratives is crucial for personal reflection as well as for collaborative knowledge building.

Set-down phase

The POST LUDUS discussion should highlight the issue of the evaluation function and its resolution.

Examples

In the Guess my Robot game (Mor et al, 2006) students exchanged challenges in the domain of number sequences. Proposers would program a ToonTalk robot to produce a sequence, keep the robot to themselves, and publish the first few terms of the sequence (Figure 10). Responders would then solve the challenge by recreating a robot to produce the same sequence and posting it as a comment on the challenge page (Figure 11). Often the response robot was different from the original, which led learners to discussions about issues such as proof and equivalence. The same structure was then used in the Guess my graph game in the domain of function graphs (Simpson, Hoyles and Noss, 2006) and the Guess my garden game in the domain of randomness and probability (Cerulli, Chioccariello and Lemut, 2007).

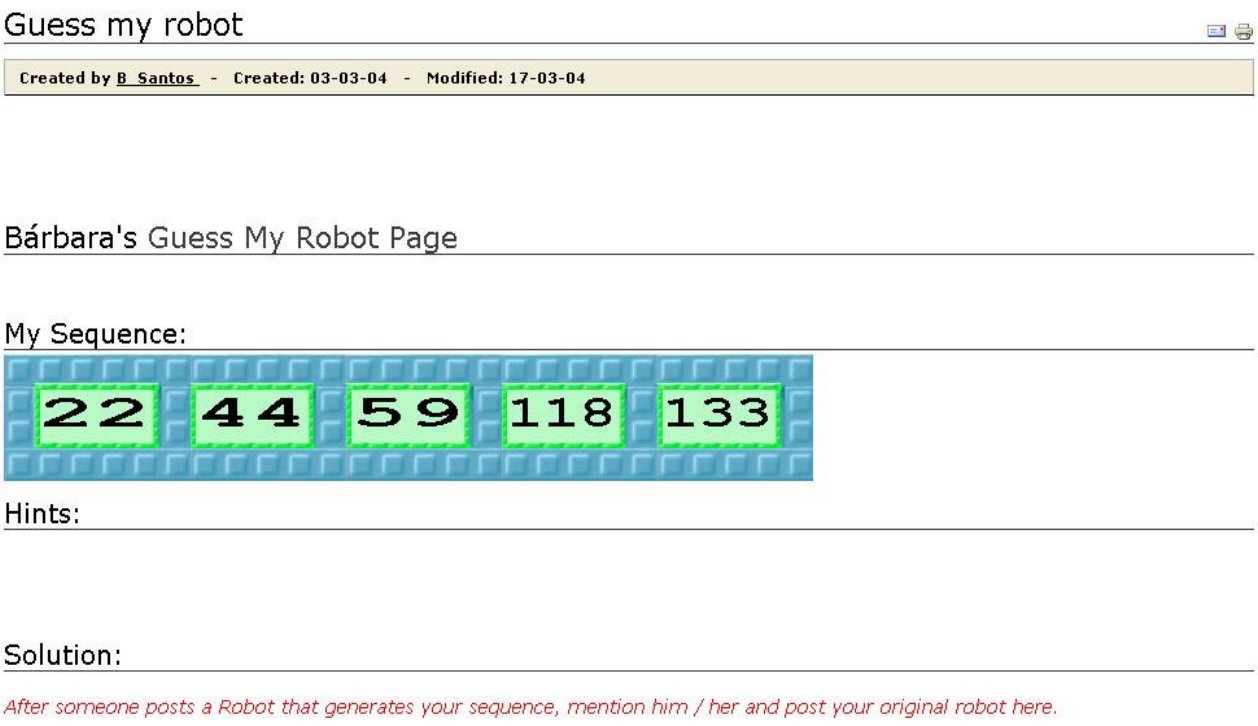


Figure 10: Example guess my robot challenge

Figure 11: Example guess my robot response

Related Patterns

Elaborates: MATHEMATICAL GAME-PIECES;

Uses: CHALLENGE EXCHANGE; UNDERCOVER PROCESS; BUILD THIS; TASK IN A BOX; OBJECTS TO TALK WITH; NARRATIVE SPACES;

Leads to: POST LUDUS;

Notes

Guess my X assumes a degree of social and technical sophistication which suggests it would be suitable for young teens and above. It can, however, be adapted for younger children.

The game requires flexibility in time to allow learning dynamics to emerge. It can be interleaved with other activities.

It is suitable for concrete, well-bounded content domains, such as computation, modelling or analysis. It uses these as a stratum for developing meta-cognitive skills of problem solving, analysis, argumentation and general mathematical discourse.

The fact that the game dynamics are driven by participants makes the educators' role subtle and critical. The educator needs to facilitate fruitful interactions, and monitor these to divert them to

high standards of mathematical discourse.

The game can be played by individuals, pairs or teams. The number and spread of participants can also vary. However, it is crucial to allow enough time for a culture to emerge. This can be achieved by interleaving the game with other activities, e.g. playing it for the last 10 minutes of each session over several weeks.

Both proposers and responders tend to converge to challenges which are HARD BUT NOT TOO HARD. When the environment encourages social cohesion, players seem to reciprocate 'good' challenges.

3. Soft scaffolding

Technology should be designed to scaffold learners' progress, but an interface that is too rigid impedes individual expression, exploration and innovation.

www.flickr.com/photos/docman/6107473

The problem

Scaffolding is a powerful tool for accelerating learning. It is a fundamental principle in many interactive learning environments, such as OISE's Knowledge Forum, and is a guiding principle in Learner-centred approaches (c.f. Quintana et al, 2004). However, scaffolds can become straitjackets when they are too imperative.

How do you provide direction and support while maintaining the learners' freedom, autonomy and sense of self, as well as the teachers' flexibility to adapt?

Forces

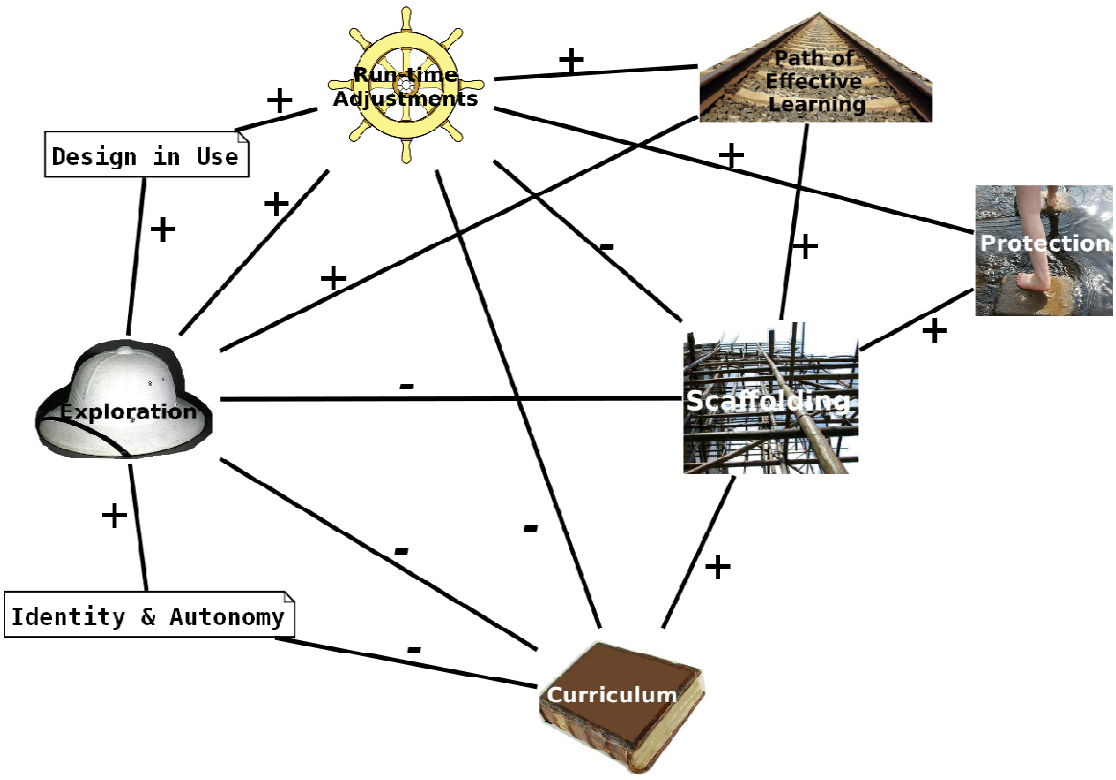


Figure 7: Force map for the Soft Scaffolding pattern

- The role of the educator, and by extension educational tools, is to direct the learner towards a productive path or enquiry.
- If the educational tool adamantly leads the learner through a set sequence, it risks failure on several accounts:
  - There is no leeway for mistakes, innovations, explorations or personal trajectories of learning.
  - Learners feel deprived of personal voice, and their motivation may falter.
  - It is hard to bypass design flaws discovered in the field or adjust to changing circumstances.

Context

Scaffolding is a term commonly used in educational design to describe structure that directs the learner's experience along an effective path of learning. This pattern originates from interactive web-based interfaces, where users can express themselves in writing. However, it should apply to almost any interactive learning interface.

Solution

Provide scaffolding which can easily be overridden by the learner or by the instructor. Let the scaffolding be a guideline, a recommendation which is easier to follow than not, but leave the choice in the hands of the learner.

- When providing a multiple-selection interface, always include an open choice, which the user can specify (select 'other' and fill in text box).
- When the user is about to stray off the desired path of activity, warn her, ask for confirmation, but do not block her.
- When providing templates for user contributions, include headings and tips but allow the user to override them with her own structure.

Examples

The ACTIVE WORKSHEETS used in the WebReports system (Mor, unpublished) provided participants a structure to work within, but allowed them to take control and change this structure as their confidence grew (Figure 8).

Explore

Can you think of a way to use your robot to produce these sequences?

Sequence	Explain
	If yes, explain how you would do it. If you think it is impossible, explain Why. Add any ToonTalk object that helps support your argument.
2, 3, 4, 5...	
-1, -2, -3, -4...	
-7, -6, -5, -4..	
2, 4, 6, 8...	
5, -1, -7...	
Write down a sequence of your own, which can be generated by your robot.	
Write down a sequence of your own, which **cannot** be generated by your robot.	

Explain

How would you explain to a friend what kind of sequences your robot can generate, and how it can be used to generate those sequences?

Describe one sequence that cannot be generated by it, and explain why.

How would you convince a friend of your claims?

Figure 8: Example of Active worksheet. Learners were given a template in which to report on their exploration, but could edit it freely and replace the structure with their own.

The ToonTalk tool packaging convention (Mor et al, 2006), which was the basis for TASK IN A BOX, prompted learners to package their own productions in a particular way by providing them with useful examples. It did not block them from developing their own packaging style, but the ToonTalk-weblabs interface did give precedence to conventionally packaged constructions (Figure 9).

Training *Add number*

Train a robot to repeatedly add 1 to produce the numbers 0, 1, 2, 3.... Call the robot *Add number*.

(Click the box to get started, post your robot here when you're done)

Figure 9: Task-in-a-box demonstrating the ToonTalk packaging convention. Learners received their tasks in the recommended style for submitting their answers, but could edit and modify it to their preference.

Related Patterns

Used by: GUESS MY X; WEBLABS PEDAGOGICAL CYCLE; NARRATIVE SPACES

Elaborated by: ACTIVE WORKSHEET

Notes

The forces of this pattern are present in face-to-face learning situations. Experienced educators resolve them by providing ADAPTIVE SUPPORT; varying the learners freedom in response to their confidence. This could be implemented by intelligent tutoring systems, but simple learning environments lack this flexibility, and tend to compensate by being over-directive.

4. Narrative spaces

Constructing narrative is a fundamental mechanism for making sense of events and observations. To leverage it, we must give learners opportunities to express themselves in narrative form.

The problem

How can the epistemic power of narrative be harnessed by educators and learners in the construction of mathematical meaning?

Forces

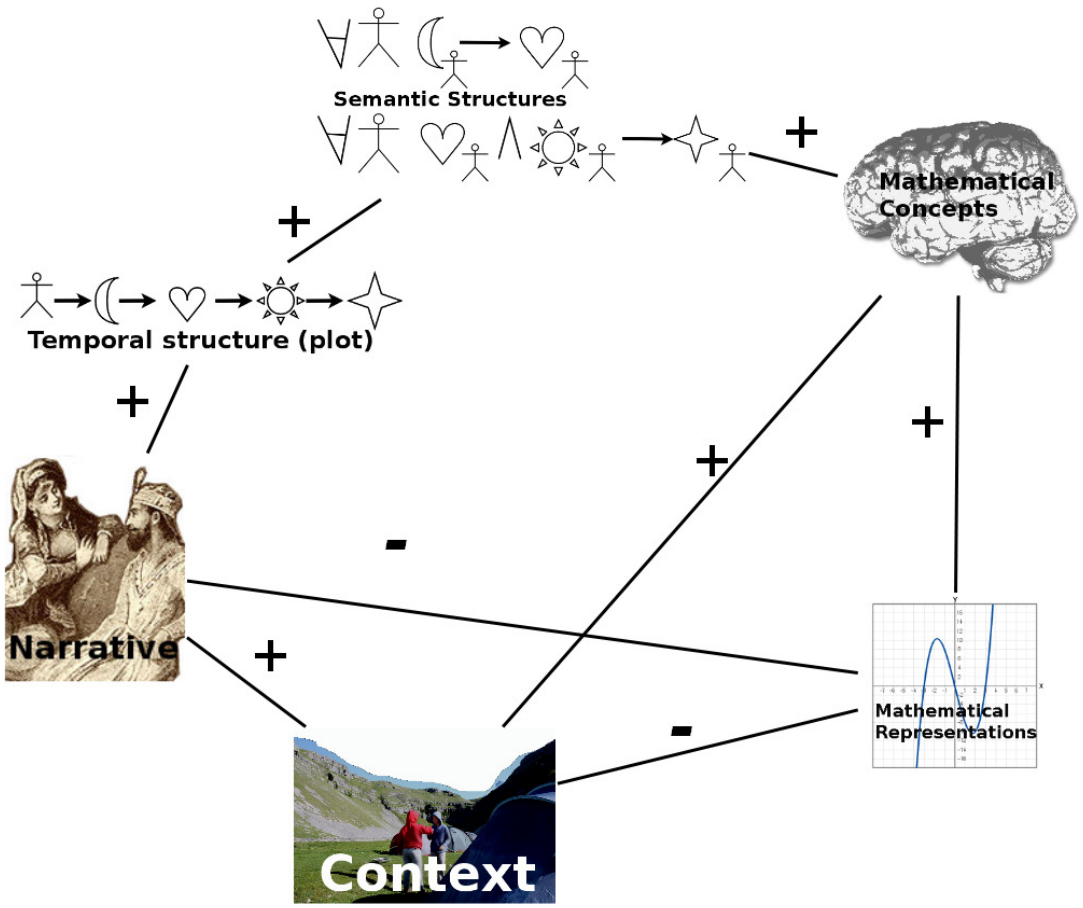


Figure 10: force map for Narrative Spaces

- Narrative is a powerful cognitive and epistemological construct (Bruner 1986; 1990; 1991).
- Mathematics appears to be antithetical to narrative form, which is always personal, contextual and time-bound.

Context

Digital environments for collaborative learning of mathematics.

Solution

Provide learners with a narrative space: a medium, integrated with the activity design, which allows learners to express and explore ideas in a narrative form:

- Allow for free-form text, e.g. by supporting SOFT SCAFFOLDING.
- Choose NARRATIVE REPRESENTATIONS when possible.

Mark narrative elements in the medium:

- Clearly mark the speaker / author, to support a sense of voice.
- Date contributions to support temporal sequentiality ('plot').
- Use SEMI-AUTOMATED META-DATA to provide context.

Examples

The webreports system allowed learners to comment on any page using a free-form WYSWYG editor (Figure 6). This allowed them to express their mathematical ideas in a personal narrative, as well as the path by which they arrived at these ideas. Using this feature, learners expressed and developed arguments which they could not yet formalize, and shared their learning process.

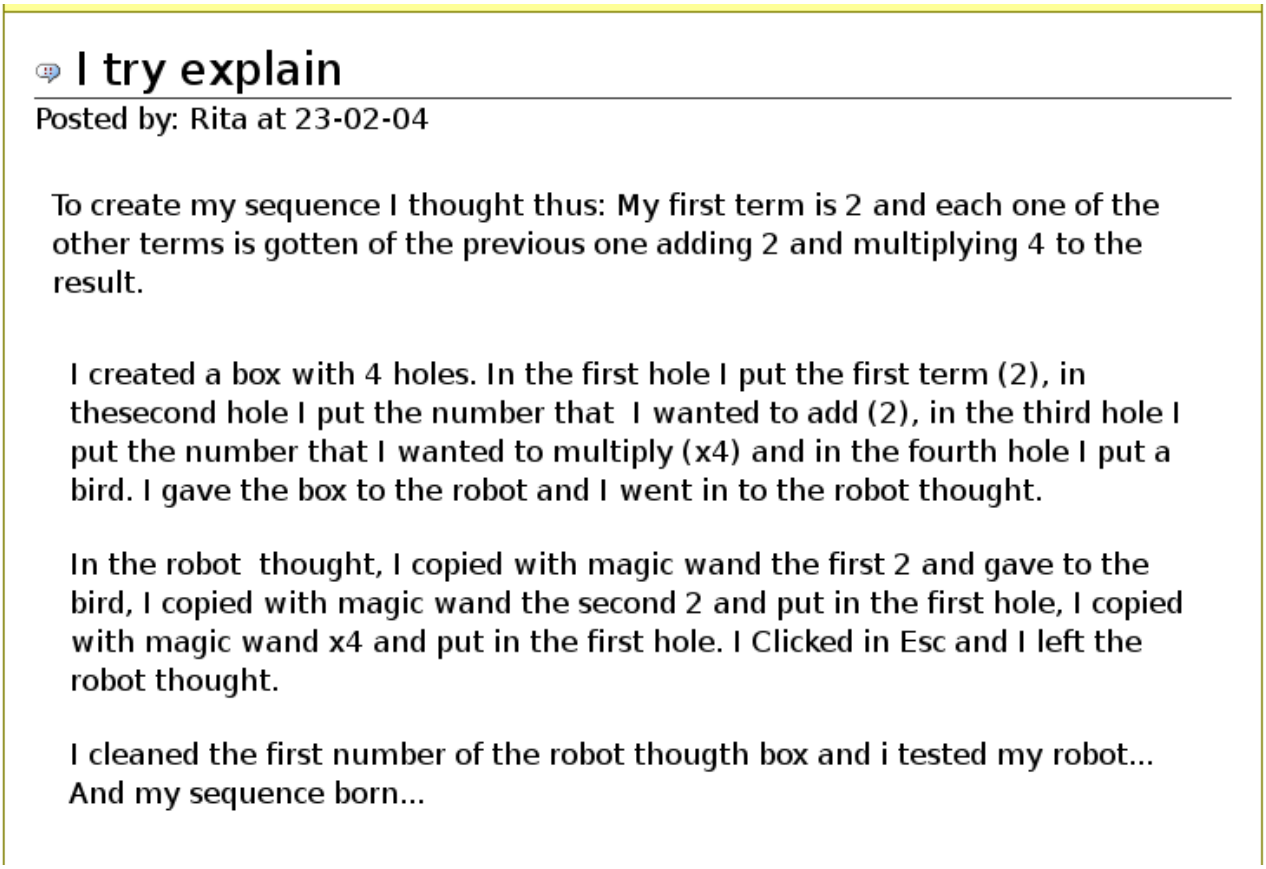


Figure 11: Example comment from the Guess my Robot game (described below). Rita expresses mathematical ideas far beyond the curriculum for her age, and shares her learning process with her peers.

One participant in the guess my robot game (described below) used ToonTalk robots in an unexpected way: he trained the robot to "act out" the way in which he solved a challenge (Figure 7).

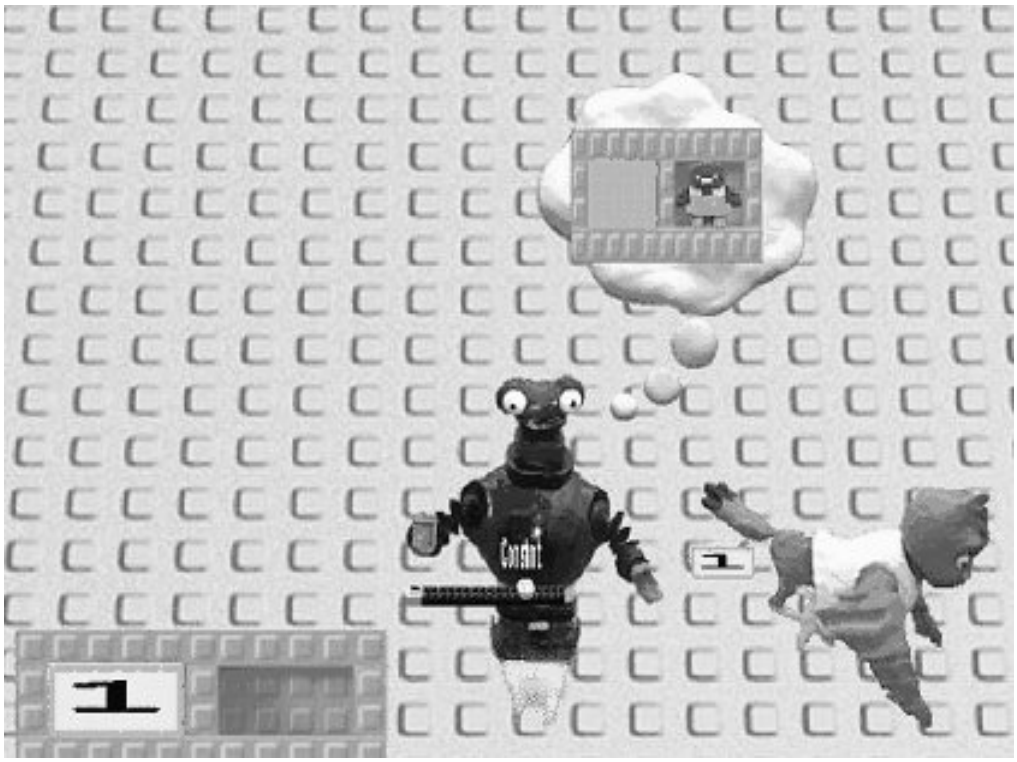


Figure 12: A robot acting out the way in which its programmer solved a mathematical challenge

Related Patterns

Uses: NARRATIVE REPRESENTATIONS; OBJECTS TO TALK WITH; SEMI-AUTOMATED META-DATA

Used by: GUESS MY X

5. Objects to talk with

Natural discourse makes extensive use of artefacts: we gesture towards objects that mediate the activity to which the discussion refers. This dimension of human interaction is often lost in computerized interfaces. When providing tools for learners to discuss their experience, allow them to easily include the objects of discussion in the discussion.

The problem

Several approaches to mathematics education highlight the importance of conversation and collaboration. The communicational approach (Sfard, 2008) equates thinking with communication, and sees learning mathematics as acquiring certain rules of discourse. Yackel and Cobb (1995) talk of the establishment of socio-mathematical norms through classroom discourse. Hurme and Järvelä (2005) argue that networked discussions can mediate students' learning, allowing students to co-regulate their thinking, use subject and metacognitive knowledge, make metacognitive judgments, perform monitoring during networked discussions and stimulates them into making their thinking visible.

Most computer-mediated discussion tools are strongly text-oriented, prompting users to express their thoughts lucidly in words or symbols. Yet two important elements of natural conversation are lost: the embodied dimension, i.e. gestures, and the ability to directly reference the objects of discussion.

Forces

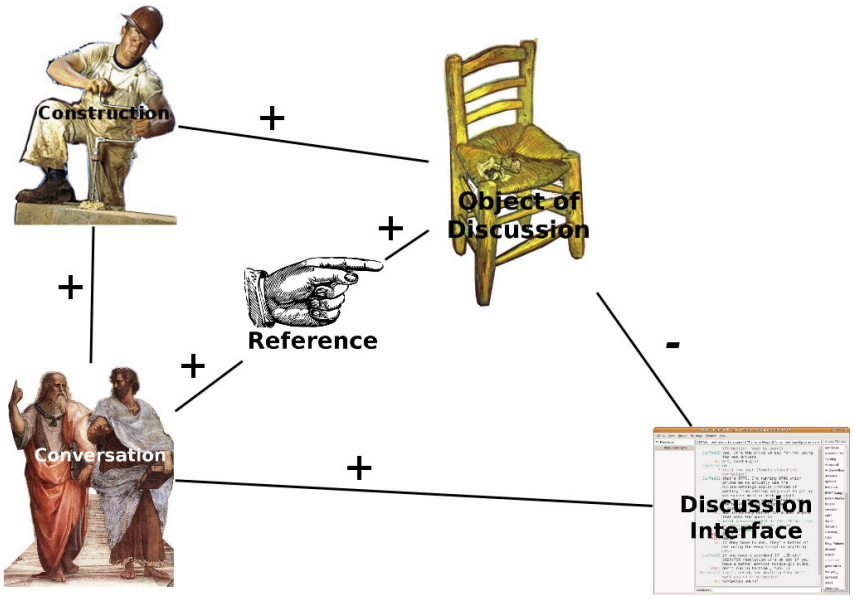


Figure 13: force map for the Objects to Talk With pattern

- Conversation is a powerful driver of learning, it:
    - Prompts learners to articulate their intuitions and in the process formulate and substantiate them.
    - Establishes mathematical norms of discourse.
    - Enables learners to share knowledge and questions.
- Conversation is even more powerful when building on personal experience or constructing or exploring mathematical objects.
- However, text based conversation media may obstruct learners, by forcing them to describe verbally the objects of enquiry which they would naturally gesture at.

Context

This pattern refers to interfaces which allow learners to converse about a common activity.

Solution

Learning activities often involve the use or construction of artefacts. When providing tools for learners to discuss their experience, allow them to easily include these artefacts in the discussion. If the activity is mediated by or aims to produce digital artefacts, then the discussion medium should allow embedding of these artefacts. The medium should support a visual (graphical, symbolic, animated or simulated) 1:1 representation of these objects.

When providing a NARRATIVE SPACE, allow the user to seamlessly embed the objects of discussion in the flow of narrative, so that learners can refer to these objects in a naturalistic manner.

In POST LUDUS discussions, the game's MATHEMATICAL GAME PIECES should become the OBJECTS TO TALK WITH. If the game is supported by a NARRATIVE SPACE, this emerges from the game flow.

Examples

This pattern identifies one of the WebReports system's primary design objectives. When developing the final version of the system, significant effort went into providing streamlined integration, which would allow students to select objects in ToonTalk and with a few clicks embed them in a webreport. The embedded objects were represented by their graphical image. When clicked, this image would invoke the original ToonTalk object in the viewers' ToonTalk environment. Likewise, when the activity involved graphs, learners could embed these in their report (Figure 14).

This is the real graph that was produced by the cumulate total of the halving-a-number robot. It looks like the top of my graph but I made the fatal mistake of thinking it started at zero. I also said it wouldn't go over 100, which was very wrong.

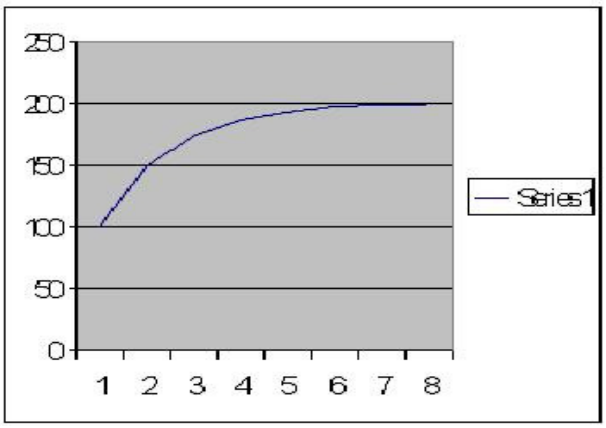


Figure 14: Webreport discussing a graphing activity, with the graph embedded as an object to talk with.

Related Patterns

Used by: GUESS MY X; TASK IN A BOX; ACTIVE WORKSHEET; SOFT SCAFFOLDING; NARRATIVE SPACES; POST LUDUS

Uses: MATHEMATICAL GAME-PIECES

Notes

The wide range of patterns which use this one indicate that it is indeed a fundamental component, applicable to most systems aiming to support discussion and collaborative learning.

Thumbnails

NARRATIVE REPRESENTATIONS	Prefer forms of representation which have narrative qualities, or afford narrative expression. These are not only textual representation, but also visual or animated forms which include elements of context, plot, and voice.
POST LUDUS	Follow a game (or any other exploratory activity) with a discussion in which participants are prompted to articulate their learning experience and acquired knowledge. Such a discussion brings intuitions to the surface, strengthening structured knowledge and exposing discrepancies.
HARD BUT NOT TOO HARD	A challenge has to be set at a level which is slightly above the participants current level. A challenge too easy will be perceived as boring, while a challenge too hard will result in frustration – both leading to disengagement.
CHALLENGE EXCHANGE	A self-regulating mechanism for implementing HARD BUT NOT TOO HARD: allow learners to set challenges for one another.
BUILD THIS	A type of challenge / game, where learners are shown an object and asked to reconstruct is.
TASK IN A BOX	When using environment A to provide tasks in environment B, package these tasks in a compact form that can be embedded in A and unpacked in B. Each task package should include the task description, any tools required to perform it, and the mechanism for reporting back the results.
UNDERCOVER PROCESS	A challenge / game / task where learners need to reverse-engineer a process by observing its effects.
SUGAR-COATING (anti-pattern)	Mathematical (or any other subject) matter is injected into a game in a disconnected way, so that the game is the sugar-coating used to help the learner swallow the bitter content. As a result the game loses its appeal, and the learner receives the message that the educational content is inherently un-enjoyable.

Conclusions

This paper presented five patterns out of an emerging language of patterns for techno-pedagogical design for mathematics learning and teaching, and outlined another seven. These design patterns reflect a techno-pedagogical approach which sees the design of educational activities and tools as a holistic endeavour. Thus, while some patterns emphasise certain features of technology and others highlight structures of activity, they all relate to some extent to both. The pedagogical approach underlying these patterns combines construction, communication and collaboration.

Above all, these patterns demonstrate the immense complexity of designing for learning. This complexity calls for further efforts towards identifying methodical frameworks for describing, aggregating and mapping design knowledge. The prime example of this complexity is the GUESS MY X pattern. At first, guess my robot may seem a simple game with surprising effects. The detailed analysis embodied in the GUESS MY X pattern, along with its 'ancestor' patterns – such as CHALLENGE EXCHANGE and BUILD THIS, suggests that the games success is not a fluke, but rather a result of an intricate assemblage of multiple design elements relating to the tools, the activity and the ways in which they interact.

Apart from the patterns themselves, this paper introduces the construct of force-maps as an innovative form supporting design pattern problem descriptions.

References

Alexander, C. W. (1964), *Notes on the Synthesis of Form*, Harvard University Press.

Bruner, J. (1991), 'The Narrative Construction of Reality', *Critical Inquiry* **18**.

Bruner, J. (1990), *Acts of Meaning : Four Lectures on Mind and Culture (Jerusalem-Harvard Lectures)*, Harvard University Press, Cambridge, MA.

Bruner, J. (1986), *Actual Minds, Possible Worlds (The Jerusalem-Harvard Lectures)*, Harvard University Press, Cambridge, MA.

Cerulli, M.; Chioccariello, A. & Lemut, E. (2007), A micoworld to implant a germ of probability, *in* ' 5th CERME conference - congress of European Society for Research in Mathematics Education, Larnaca, Cyprus'.

Disessa, A. A. & Cobb, P. (2004), 'Ontological Innovation and the Role of Theory in Design

Experiments', *Journal of the Learning Sciences* **13**(1), 77-103.

Folcher, V. (2003), 'Appropriating artifacts as instruments: when design-for-use meets design-in-use', *Interacting with Computers* **15**(5), 647-663.

Hurme, T. & Järvelä, S. (2005), 'Students' Activity in Computer-Supported Collaborative Problem Solving in Mathematics', *Journal International Journal of Computers for Mathematical Learning* **10**(1), 49--73.

Kahn, K. (2004), The child-engineering of arithmetic in ToonTalk, *in* 'IDC '04: Proceeding of the 2004 conference on Interaction design and children', ACM Press, New York, NY, pp. 141-142.

Kahn, K. (1996), 'ToonTalk - An Animated Programming Environment for Children', *Journal of Visual Languages and Computing* **7**(2), 197-217.

Morgado, L. & Kahn, K. (in press), 'Towards a specification of the ToonTalk language', *Journal of Visual Languages and Computing*.

Mor, Y. & Noss, R. (2008), 'Programming as Mathematical Narrative', *International Journal of Continuing Engineering Education and Life-Long Learning (IJCEELL)* **18**(2), 214-233.

Mor, Y. & Winters, N. (2007), 'Design approaches in technology enhanced learning', *Interactive Learning Environments* **15**(1), 61-75.

Mor, Y.; Noss, R.; Hoyles, C.; Kahn, K. & Simpson, G. (2006), 'Designing to see and share structure in number sequences', *the International Journal for Technology in Mathematics Education* **13**(2), 65-78.

Papert, S. & Harel, I. (1991), Situating Constructionism, *in* Seymour Papert & Idit Harel, ed.,'Constructionism', Ablex Publishing Corporation, Norwood, NJ.

Quintana, C.; Soloway, E. & Norris, C. A. (2001), Learner-Centered Design: Developing Software That Scaffolds Learning., *in* 'ICALT', pp. 499-500.

Rabardel, P. & Bourmaud, G. (2003), 'From computer to instrument system: a developmental perspective', *Interacting with Computers* **15**(5), 665 - 691.

Ruthven, K. (2008), 'The Interpretative Flexibility, Instrumental Evolution and Institutional Adoption of Mathematical Software in Educational Practice: The Examples of Computer Algebra and Dynamic Geometry', *Journal of Educational Computing Research* **39**(4), 379-394.

Sfard, A. (2008), *Thinking as Communicating: Human Development, the Growth of Discourses, and Mathematizing*, Cambridge University Press.

Simpson, G.; Hoyles, C. & Noss, R. (2005), 'Designing a programming-based approach for modelling scientific phenomena', *Journal of Computer Assisted Learning* **21**(2), 143-158.

Simon, H. A. (1996), *The Sciences of the Artificial - 3rd Edition*, The MIT Press, Cambridge, MA.

Trouche, L. (2005), 'Instrumental genesis, individual and social aspects', *The didactical challenge of symbolic calculators: Turning a computational device into a mathematical instrument*, 197-230.

Trouche, L. (2003), 'From artifact to instrument: mathematics teaching mediated by symbolic calculators', *Interacting with Computers* **15**(6), 783-800.

Winters, N. & Mor, Y. (2008), 'IDR: a participatory methodology for interdisciplinary design in technology enhanced learning', *Computers and Education* **50**(2), 579-600.

Yackel, E. & Cobb, P. (1995), Classroom Sociomathematical Norms and Intellectual Autonomy, *in* 'Nineteenth International Conference for the Psychology of Mathematics Education', Program Committee of the 19th PME Conference, Recife, Brazil, pp. 264-271.

"Choose Your Own Architecture"

Interactive Pattern Storytelling

James Siddle, IBM UK Ltd, james.siddle@uk.ibm.com
Illustrations by Maisie Platts, maisie_platts@yahoo.co.uk

"You peer into the gloom to see dark, slimy walls with pools of water on the stone floor in front of you. The air is cold and dank. You light your lantern and step warily into the blackness. Cobwebs brush your face and you hear the scurrying of tiny feet: rats most likely. You set off into the cave. After a few yards you arrive at a junction.
Will you turn west (turn to 71) or east (turn to 278)?"

The Warlock of Firetop Mountain (1) [Jackson+]

Introduction

This paper proposes making pattern stories interactive, in order to be more engaging, educational, and fun, and to support the exploration of pattern-based designs. The *"Choose Your Own Adventure"* [CYOA1] style of book is suggested as a suitable basis for introducing interactivity into pattern stories.

Two interactive pattern stories appear in this paper. These stories are based on a story describing "A Request Handling Framework" that appears in *"Pattern Oriented Software Architecture, Volume 5: On Patterns and Pattern Languages"* [POSA5]. The first story is interactive around design alternatives, and illustrates different consequences that can occur given different design choices. The second story is interactive around requirements, allowing the reader to choose problems they want to solve, illustrating how pattern languages can solve a variety of related problems. Both stories provide the reader with the opportunity to explore pattern-based designs.

The rest of this paper is structured as follows. The target audience is introduced first, followed by brief introduction to pattern and interactive fiction concepts. The origin and structure of the two interactive stories is then described, and a reader guidance section provides essential information needed to read the stories. This is followed by the interactive stories themselves. The paper closes with an analysis of story features, a comparison between the stories presented, and a brief discussion of the benefits, liabilities, and applicability of the approach.

Proceedings of the 13th European Conference on Pattern Languages of Programs (EuroPLoP 2008), edited by Till Schümmer and Allan Kelly, ISSN 1613-0073 <issn-1613-0073.html>.

Target Audience

By reading this paper, computer science students, software developers, and software architects will gain an insight into the application of patterns, the choices available during software design, and will learn several desirable and undesirable design choices in the context of request handling. Patterns theorists and authors will learn how to combine interactive fiction and pattern concepts to create interactive pattern stories for patterns-based education. Technical writers may also benefit from learning an interactive approach to describing the design and development of software through patterns.

Concepts

Patterns, pattern stories, pattern languages

A *pattern* [Alexander+77] is a solution to a problem that occurs in a particular context, captured in an easy to understand format. A *pattern story* [Henney06] describes the application of one or more patterns. Pattern stories can be derived from *pattern languages* [Alexander+77], which connect patterns together to provide guidance in solving wider problems than is possible with individual patterns. A key feature of pattern languages is that patterns are connected together via a shared context, where the application of one pattern creates a context in which another pattern can be applied.

To apply a pattern language, one follows the connections in the language to build up a *sequence* [Henney06] of patterns. Each pattern application solves one part of the overall problem, after which the reader determines the next sub-problem they want to tackle (the pattern texts can help with this). The reader then follows a connection from one of the patterns they have already applied, to solve the next part of the overall problem. This continues until the reader's overall problem has been fully solved, or the pattern language is unable to help the reader further.

It should be noted however that patterns and the associated structures, concepts and approaches are not a silver bullet for designing software – for example a pattern or pattern language may only cover part of the problem space for a given context, which may leave the designer with a partial solution. The quality of a design derived from a pattern language is dependent on how extensive and rich the language is. Additionally, effective use of patterns relies on the designer treating them as design guidance rather than prescriptive solutions; the designer must use their knowledge of the specific problem being faced to fill in the gaps in any particular pattern.

"Choose Your Own Adventure" and Interactive Fiction

"Choose Your Own Adventure" [CYOA1] books are a form of children's literature which is interactive in nature. The reader typically starts at a single entry point which describes the overall context for the story, then is presented with several decisions each of which lead to further story, and further decision points, etc. Eventually the reader will come to one of the many endings, some of which will be good, others bad. A short history of interactive fiction can be found in the first appendix.

Interactive Pattern Stories

Origin of the stories presented

This paper presents interactive stories that are based heavily on a pattern story published in *"Pattern-Oriented Software Architecture, Volume 5: On Patterns and Pattern Languages"*[1] [POSA5]. In this story, a collection of patterns are applied to create a framework for handling requests. Various problems are posed, such as how to encapsulate or uniformly handle requests, and various patterns are applied (in the story) to solve the problems. This pattern story was originally derived from the pattern language published in [POSA4].

Rather than write a completely new pattern story from scratch, the request handling framework story is taken (almost verbatim), and transformed into the interactive versions that appear below. The text is reworded into second-person (a defining characteristic of interactive fiction stories), and decisions and associated consequences are introduced into the tail end of the story. Specifically, the reader is able to make decisions relating to the STRATEGY, TEMPLATE METHOD, NULL OBJECT, and COMPOSITE COMMAND patterns.

The design alternatives and consequences in the stories were derived from variations to the requesting handling framework story which are presented in [POSA5], as well as the pattern language in [POSA4].

Story structure

Two variations of the interactive pattern-story format are presented - one provides the reader with design choices, the other with choices related to functional requirement fulfilment.

The first story allows the reader to explore the consequences of applying different patterns to fulfil a fixed set of functional requirements, which are a system's capabilities, services, and behaviour [Bass+03]. This story allows the reader to experience the consequences of choosing both optimal and sub-optimal design alternatives. The consequences of the reader's choices are described in terms of quality requirements -also known as *'-ilities'*, these are the qualities of the system being developed that are influenced by design decisions taken [Bass+03]. The first story has been enhanced with illustrations to show some of the possible 'real world' consequences of the decisions in the story.

The second interactive story variation allows the reader to choose which functional requirements they wish to fulfil, and illustrates how a collection of related problems can be solved by applying a pattern language. In this story, there is only one possible way to fulfil any particular functional requirement.

---

[1] Frank Buschmann, Kevlin Henney, Douglas C. Schmidt. © 2007, John Wiley & Sons Limited. Reproduced with permission.

Guidance on Reading the Stories

Before proceeding to the interactive stories, it's important to understand who should read them, the requirements around which they are based, and how to read them. This information is presented as reader guidance below.

Who should read the stories

Computer science students, software developers, software architects, technical writers, patterns theorists and pattern authors will benefit from reading the stories. The reader is referred to the target audience section at the start of the paper for audience motivation statements, in order to avoid unnecessary duplication of information.

Requirements

The interactive stories that appear below are based around certain requirements. There are two functional requirements, and three quality requirements. The functional requirements serve as the basis of reader choices, whilst the quality requirements are expected of the framework being developed in the stories, and of any concrete systems that are built on top of it.

The specific role of these requirements to the interactive stories will be made clear in a short introductory paragraph before each story, and you may wish to refer back to this point when you are presented with choices in the stories.

Functional requirements:

F1. Support for an optional logging policy mechanism to allow requests that are handled by the framework to be logged in a variety of ways. This mechanism is expected to be used to allow different qualities of service (such as the level of detail provided) for different deployments of the request handling framework.

F2. The ability to create compound requests, to support composition of commands that have been written to be processed by the framework.

Quality requirements:

Q1. Developers and users of the framework should find it easy to work with (*understandability*),

Q2. It should be easy to perform routine maintenance of framework and framework-using code, such as fault correction or performance improvement (*maintainability*),

Q3. It should also be easy to take advantage of new software or hardware technologies that may become available in the future (*evolvability*).

The requirements are referred to throughout the interactive stories via the unique codes above (e.g. Q1 denoting quality requirement number one, understandability).

How to read the stories
Start reading at step 1 which appears in the next section. Step 1 describes a context that is shared by both stories[2], then presents the choice of which story variation to read. "Varying Design Choices" appears first, and is followed by "Varying Requirements".

Make sure you read the introduction to each story, and then simply follow the decision instructions as they appear. Route maps for both stories can be found in the appendices, along with thumbnails for each pattern used.

Here are a few other things to bear in mind whilst reading the stories:

- The decisions presented are intentionally short on information in order to keep each story succinct, and to promote exploration of the design space. Under ideal circumstances, design decisions would be made based on an assessment of all relevant information, but this is rarely the case on real projects so the decisions do represent realistic choices.

- A valid option at each decision point is to go back a step – after all most software projects employ some form of source control, allowing earlier versions of source code to be reverted to. The reader is asked to take this as an implicit option, which simplifies the presentation of available options.

- Similarly, the choices presented do not represent the entire set of decisions available, rather a subset chosen in order to explore software design and development in the particular context. In reality, a software professional is always free to make whatever choice they wish. More experienced or advanced practitioners may find the choice constraints limiting.

---

[2] In addition to those listed above, other functional requirements apply to step 1. These are not explicitly listed to ensure the information presented is relevant to the interactive portions of the stories.

The Interactive Stories

Context

1

You are developing an extensible request-handling framework for your system, and are faced with the problem of how requests can be issued and handled so that the request handling framework can manipulate the requests explicitly.

You decide to objectify requests as COMMAND objects, based on a common interface of methods for executing client requests. COMMAND types can be expressed within a class hierarchy, and clients of the system can issue specific requests by instantiating concrete COMMAND classes and calling the execution interface. This object can then perform the requested operations on the application and return the results, if any, to the client.

The language chosen for implementing the framework is statically typed, and there may be some implementation common to many (or even all) COMMANDs in your system. You wonder what the best form for the COMMAND class hierarchy is.

You decide to express the root of the hierarchy as an EXPLICIT INTERFACE. Both the framework and clients can treat it as a stable and published interface in it's own right, decoupled from implementation decisions that affect the rest of the hierarchy. You decide that concrete COMMAND classes will implement the root EXPLICIT INTERFACE, that common code can be expressed in abstract classes below the EXPLICIT INTERFACE rather than in the hierarchy root, and that concrete classes are expressed as leaves in the hierarchy.

You realise that there may be multiple clients of a system that can issue COMMANDs independently, and wonder how COMMAND handling can be handled generally.

You decide to implement a COMMAND PROCESSOR to provide a central management component to which clients pass their COMMAND objects for further handling and execution. The COMMAND PROCESSOR depends only on the EXPLICIT INTERFACE of the COMMAND hierarchy.

You also realise that the COMMAND PROCESSOR makes it easy to introduce a rollback facility, so that actions performed in response to requests can be undone. You extend the EXPLICIT INTERFACE of the COMMAND with the declaration of an undo method (which will affect the concreteness of any implementing classes), and decide that the COMMAND PROCESSOR will handle the management.

After introducing the undo mechanism, you recognise that there is also a need for a redo facility, to allow previously undone COMMAND objects to be re-executed. You need to determine how the COMMAND PROCESSOR can best accommodate both undo history and redo futures for COMMAND objects.

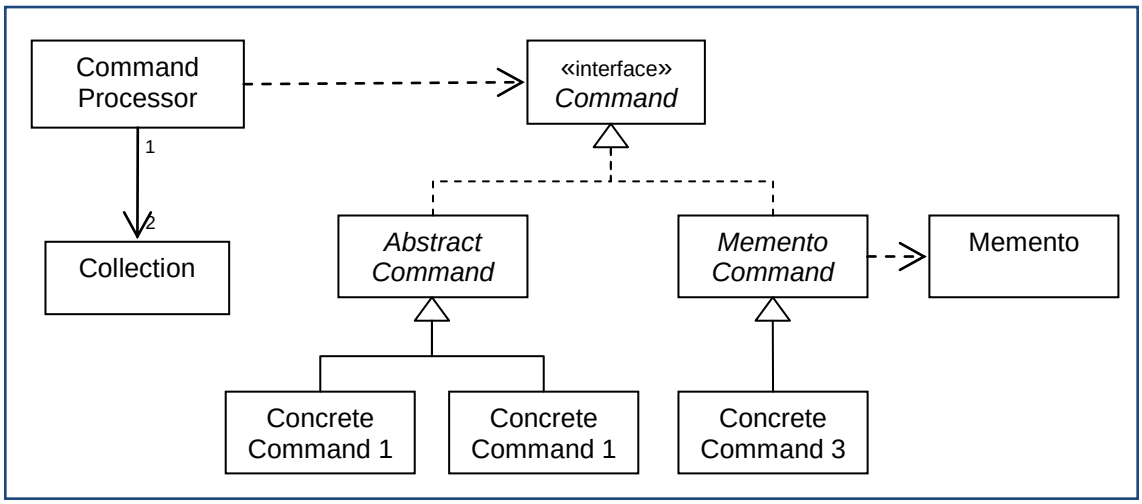
You decide to add COLLECTIONS FOR STATES to the COMMAND PROCESSOR, so that one collection holds COMMAND objects that have already been executed – and can

therefore be undone – while another collection holds COMMAND objects that have already been undone – and can therefore be re-executed. You make both collections into sequences with 'last in, first out' stack-ordered access.

You understand that some actions may be undone (or redone) quite simply, but that others may involve significant state changes that complicate a rollback (or rollforward). You wonder how the need for a simple and uniform rollback mechanism can be balanced with the need to deal with actions that are neither simple nor consistent with other actions.

You decide to allow COMMAND objects to be optionally associated with MEMENTOs that maintain whole or partial copies of the relevant application state, as it was before the COMMAND was executed. You also decide that those COMMAND types that require a MEMENTO will share common structure and behaviour for setting and working with the MEMENTO's state. You express this commonality by introducing an abstract class that in turn implements the COMMAND's EXPLICIT INTERFACE; MEMENTO based COMMAND types can then extend this abstract class. COMMAND types that are not MEMENTO based won't inherit from this abstract class, implementing the EXPLICIT INTERFACE directly, or extending another abstract class suitable for their purpose.

The following UML diagram shows the story so far:



The story continues in both "Varying Design Choices" on page 8, and "Varying Requirements" on page 15.

Story 1 - Varying Design Choices

In the following interactive story, you fulfil all of the functional requirements that were introduced above. The choices you make in the following story determine to what extent you fulfil the quality requirements or not, and the text of the story describes the consequences of your design decisions in relation to those quality requirements.

Now continue at step 2...

2

You now realise that the framework needs a logging facility for requests, and wonder how logging functionality can be parameterized so that users of the framework can choose how they wish to handle logging, rather than the logging facility being hard-wired.

If you wish to use inheritance to support variations in housekeeping functionality, turn to 7.

Otherwise if you prefer the use of delegation, turn to 3.

3

You choose to express logging functionality as a STRATEGY of the COMMAND PROCESSOR, so that a client of the framework can select how they want requests logged by providing a suitable implementation of the STRATEGY interface. This ensures that the common COMMAND PROCESSOR behavioural core is encapsulated in one class, while variations in logging policy are separated into other classes, each of which implements the STRATEGY interface.

Clients of the request handling framework can select how they want logging performed by choosing which STRATEGY to instantiate the COMMAND PROCESSOR with. Some users will want to just use the standard logging options, while others may wish to define their own custom logging, so you ensure the framework provides some predefined logging types.

This clean separation supports the understandability (Q1), maintainability (Q2), and evolvability (Q3) of both the framework and any additional logging policy classes introduced as part of concrete deployments.

Having introduced a parameterized logging facility, you wonder how the optionality of logging can be realised, in the knowledge that it makes little functional difference to the running of the framework.

> *If you wish to make changes to the COMMAND PROCESSOR control flow to take account of optionality, turn to* 8.
>
> *Otherwise if you prefer a more transparent solution, turn to* 4.

4

You provide a NULL OBJECT implementation of the logging STRATEGY which doesn't do anything when it is invoked, but uses the same interface as the operational logging implementations. This selection through polymorphism ensures that you don't need to introduce difficult to understand control flow selection within the framework to accommodate the optional behaviour, and ensures understandable (Q1) and maintainable (Q2) framework code.

Turn to 5.

5

Your request handling framework is almost complete; but you still need to ensure that compound requests are handled. Compound requests correspond to multiple requests performed in sequence and as one; they are similarly undone as one. The issue you face is how compound requests can be expressed without upsetting the simple and uniform treatment of COMMANDs within the existing infrastructure.

If you want to create a special kind of COMMAND to deal with all compound requests, turn to 6.

Otherwise, if you're happy for compound requests to be handled by the framework as it stands, turn to 9.

6

You decide to implement a compound request as a COMPOSITE COMMAND object that aggregates other COMMAND objects. To initialise a COMPOSITE COMMAND object correctly, you ensure that other COMMAND objects (whether primitive or COMPOSITE themselves) must be added to it in sequence.

This special type of COMMAND enables arbitrary compound requests to be created and composed, simplifying use of the request handling framework and avoiding the need for complex, tightly coupled, dedicated compound request classes - enhancing the maintainability (Q2) and evolvability (Q3) of client code. This comes at the cost, however, of a reduction in the understandability (Q1) of framework code – COMPOSITE implementations can be complex and non-obvious.

Turn to 10.

6 – An evolvable design supports the easy addition of new features, for example the addition of SMS based delivery notifications to an online shopping service

7

You decide to introduce a logging TEMPLATE METHOD to the COMMAND PROCESSOR class, then call the abstract method whenever logging is required within the COMMAND PROCESSOR. By necessity, you make the COMMAND PROCESSOR class abstract.

Different logging policies are provided by creating subclasses of the COMMAND PROCESSOR. This ensures that the common COMMAND PROCESSOR behavioural core is encapsulated in a superclass, while variations in logging policy are separated into different classes, each of which implements the TEMPLATE METHOD. Clients of the request handling framework can select how (or if) they want logging performed by choosing which subclass to instantiate. Some users will want to just use the standard logging options, while others may wish to define their own custom logging, so you ensure the framework provides some predefined logging subclasses.

This clean separation supports the understandability (Q1), maintainability (Q2), and evolvability (Q3) of both the framework and any additional logging policy classes introduced as part of concrete deployments.

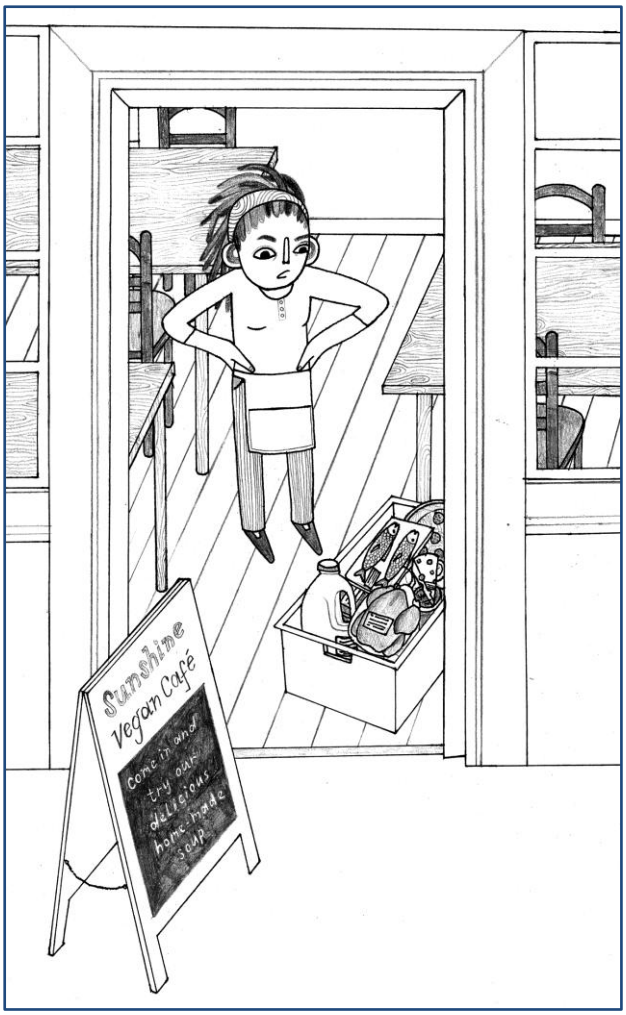
Turn to 5.

8

You decide to branch explicitly whenever a null logging STRATEGY object reference is detected within the COMMAND PROCESSOR.

Unfortunately this introduces a great deal of repetition and complexity into the class, reducing understandability (Q1) and maintainability (Q2) of the framework code. A knock-on effect of this may even be a reduction in system reliability, if, for example, checks for null object references are forgotten.

Turn to 5.



8 – An unexpected null pointer exception may leave a system in an inconsistent state, causing an online shopping system to send an order to the wrong person.

9

You decide to support compound requests through concrete COMMAND objects which aggregate other COMMAND objects. You don't need to make any changes to the existing framework because this type of functionality is already supported. But while this decision means the request handling framework itself is simpler, supporting understandability (Q1) and maintainability (Q2) of framework code, it means that clients of the framework will find it harder to use. Clients will need to represent each different compound request via a unique concrete class, which will be difficult to maintain (Q2), and harder to evolve (Q3).

Turn to 10.

10

Congratulations, your request handling framework is complete! You've introduced an optional logging policy mechanism and support for compound requests. But is it easy to use, and is it easy to maintain? Is it everything you'd hoped for? The decisions were yours, so whatever they were, you now have to deal with the consequences!

The End

Story 2 - Varying Requirements

In the following interactive story, you decide the make-up of your system by choosing which functional requirements to fulfil at each step. The quality requirements also apply to this story, but no choices related to quality requirements are available.

Now continue at step 2...

2

You now realise that the framework might benefit from a logging facility for requests, and wonder how logging functionality can be parameterized so that users of the framework can choose how they wish to handle logging, rather than the logging facility being hard-wired.

If you wish to introduce a logging facility into the framework, turn to 3.

Otherwise, if you're happy to leave the framework without a logging facility, turn to 6.

3

You choose to express logging functionality as a STRATEGY of the COMMAND PROCESSOR, so that a client of the framework can select how they want requests logged by providing a suitable implementation of the STRATEGY interface. This ensures that the common COMMAND PROCESSOR behavioural core is encapsulated in one class, while variations in logging policy are separated into other classes, each of which implements the STRATEGY interface.

Clients of the request handling framework can select how they want logging performed by choosing which STRATEGY to instantiate the COMMAND PROCESSOR with. Some users will want to just use the standard logging options, while others may wish to define their own custom logging, so you ensure the framework provides some predefined logging types.

This clean separation supports the understandability (Q1), maintainability (Q2), and evolvability (Q3) of both the framework and any additional logging policy classes introduced as part of concrete deployments.

Having introduced a parameterized logging facility, you wonder how the optionality of logging can be realised, in the knowledge that it makes little functional difference to the running of the framework.

> *If you wish to introduce transparent handling of situations when there is no logging strategy, turn to 4.*
>
> *Or if you don't think the framework needs special handing for this situation, turn to 10.*

4

You provide a NULL OBJECT implementation of the logging STRATEGY which doesn't do anything when it is invoked, but uses the same interface as the operational logging implementations. This selection through polymorphism ensures that you don't need to introduce difficult to understand control flow selection within the framework to accommodate the optional behaviour, and ensures understandable (Q1) and maintainable (Q2) framework code.

Your request handling framework is almost complete; but you wonder if you need to ensure that compound requests are handled. Compound requests correspond to multiple requests performed in sequence and as one; they are similarly undone as one. The issue you would face is how compound requests can be expressed without upsetting the simple and uniform treatment of COMMANDs within the existing infrastructure.

> *If you want to create a special kind of COMMAND to deal with all compound requests, turn to* 5.
>
> *Otherwise, if you're not worried about compound request handling , turn to* 13.

5

You decide to implement a COMPOSITE COMMAND ... go to step 14 to see what happens.

Congratulations, your request handling framework is complete! You've successfully introduced support for a logging facility, which in addition to allowing transparent policy selection can also be seamlessly disabled via a special 'null' strategy object. In addition to that, your framework also supports uniform handling of both individual and compound requests, via a special 'composite' command object. By using this object, clients of the framework will avoid introducing their own compound request objects, which can be complicated, fragile, and difficult to maintain.

The End

6

Your request handling framework is almost complete; but you wonder if you need to ensure that compound requests are handled. Compound requests correspond to multiple requests performed in sequence and as one; they are similarly undone as one. The issue you would face is how compound requests can be expressed without upsetting the simple and uniform treatment of COMMANDs within the existing infrastructure.

If you want to create a special kind of COMMAND to deal with all compound requests, turn to 7.

Otherwise, if you're not worried about compound request handling , turn to 8.

7

You decide to implement a COMPOSITE COMMAND ... go to step 14 to see what happens.

Congratulations, your request handling framework is complete! Your framework now supports uniform handling of both individual and compound requests, via a special 'composite' command object. By using this object, clients of the framework will avoid introducing their own compound request objects, which can be complicated, fragile, and difficult to maintain.

The End

8

Congratulations, your request handling framework is complete! It was already perfect for your needs, so you've decided to leave it just as it is.

The End

9

You exclaim 'xyzzy!'. You are spontaneously transported 100 miles away into the middle of the countryside, where you discover your true calling as a druid and spend the rest of your life living in a swamp.

The End

10

Your request handling framework is almost complete; but you wonder if you need to ensure that compound requests are handled. Compound requests correspond to multiple requests performed in sequence and as one; they are similarly undone as one. The issue you would face is how compound requests can be expressed without upsetting the simple and uniform treatment of COMMANDs within the existing infrastructure.

> *If you want to create a special kind of COMMAND to deal with all compound requests, turn to* 11.
>
> *Otherwise, if you're not worried about compound request handling , turn to* 12.

11

You decide to implement a COMPOSITE COMMAND ... go to step 14 to see what happens.

Congratulations, your request handling framework is complete! You've successfully introduced support for a logging facility which allows transparent policy selection. In addition to that, your framework also supports uniform handling of both individual and compound requests, via a special 'composite' command object. By using this object, clients of the framework will avoid introducing their own compound request objects, which can be complicated, fragile, and difficult to maintain.

The End

12

Congratulations, your request handling framework is complete! You've successfully introduced support for a logging facility which allows transparent policy selection.

The End

13

Congratulations, your request handling framework is complete! You've successfully introduced support for a logging facility, which in addition to allowing transparent policy selection can also be seamlessly disabled via a special 'null' strategy object.

The End

14

You decide to implement a compound request as a COMPOSITE COMMAND object that aggregates other COMMAND objects. To initialise a COMPOSITE COMMAND object correctly, you ensure that other COMMAND objects (whether primitive or COMPOSITE themselves) must be added to it in sequence.

This special type of COMMAND enables arbitrary compound requests to be created and composed, simplifying the use of the request handling framework and avoiding the need for complex, tightly coupled, dedicated compound request classes - enhancing the maintainability (Q2) and evolvability (Q3) of client code. This comes at the cost, however of a reduction in the understandability (Q1) of framework code – COMPOSITE implementations can be complex and non-obvious.

Now return to the step that sent you here...

Analysis

The stories presented above allow a reader to explore the different designs that can be derived from a pattern language within a particular context, along with the negative consequences that can come from non-pattern-based solutions in that context. Below, the features of the stories that were presented are discussed, the two stories are compared, then the benefits and liabilities of the approach are examined.

Interactive Story Features

Alternative decision points

In the first story, one design alternative allows the choice of differing but equally desirable solutions to problems. At step 2, the reader's choice leads to either TEMPLATE METHOD or STRATEGY, both reasonable solutions given the context.

Optimal versus Sub-optimal decision points

The first story also allows the reader to explore the negative consequences that may be encountered if the desirable solution for the context (i.e. pattern) is not selected. For example at Step 3, the reader either opts for a transparent solution which leads to NULL OBJECT, or to introduce complicated control flow to deal with a missing STRATEGY.

Optional requirement decision points

The second story is focused on fulfilling requirements, so the decisions allow the reader to select which functional requirements they are interested in fulfilling. For example at step 2, the reader can choose not to introduce a transparent logging policy, so the remaining decision for this route at step 6 only discusses support for compound requests.

Joining branches

In "Varying Design Choices", the story branches but the narrative is rejoined in two places, at steps 5 and 10. This demonstrates that not all branches in the story are irreconcilable. The story can be rejoined at these two points because the context of the remaining story from step 5, and at step 10, is unaffected by differences between the branches.

Specifically, the choice of how to support compound requests at step 5 is unaffected by the choice of logging policy mechanism that was made previously. Similarly, the ending of the story at step 10 is intentionally vague and unrelated to the specific design choices taken; this is only possible because the consequences of each decision are described along the way. As such the ending could be either desirable or undesirable, and this depends on the consequences the reader has built up as they have gone. In this case, the journey really is more important than the destination.

Story summaries

In the "Varying Requirements" story, there are exactly six (or is it seven?) endings, each of which provides a brief summary of the final request handling framework that the reader has chosen. This is only possible because no branches have joined in this story.

Shared descriptions

Many of the story 'nodes' in "Varying Requirements" are similar (see 5, 7 and 11) but slightly different because of the different context that occurs in each case – a fully branching story may contain many similar nodes. As such, each of steps 5, 7 and 11 describe the introduction of COMPOSITE COMMAND in a way that helps prevent redundancy in the story text.

Effectively the detailed description of the design, implementation, and consequences associated with the reader's decision are separated into a new paragraph. This is 'called' from several places, and the shared step directs the reader to return to the originating step at the end. A useful metaphor for understanding this mechanism is that of the sub-procedure (in structured programming terms), or method (in object oriented terms).

Illustrations

Finally, the "Varying Design Choices" story includes a number of illustrations associated with particular story steps. These act to tie the decisions made by the reader to 'real world' consequences, both to illustrate the possible consequences of the reader's decisions, and also to make the story more engaging.

Story Comparison

To compare and contrast the different stories available to the reader in the two variations, consider the following routes:

Varying Design Choices

Route ‹1,2,3,4,5,6,10›: The reader selects a delegation approach to introducing logging policy (i.e. STRATEGY), a transparent mechanism for handling a missing logging policies (i.e. NULL OBJECT), and a special COMMAND object for handling compound requests (i.e. COMPOSITE COMMAND).

Route ‹1,2,3,8,5,9,10›: The reader selects a delegation approach to introducing logging policy (i.e. STRATEGY), but chooses to introduce special control flow handling for missing STRATEGY objects, and to ignore special handling of compound requests.

The difference between the two routes should be clear – the former route takes all possible optimal choices, while the latter takes all possible sub-optimal choices. In both cases, the choice of STRATEGY is a neutral choice because the alternative was equally viable.

Varying Requirements

Route ‹1,2,3,4,5›: The reader selects to introduce a logging policy, transparent handling of a missing logging policies, and a mechanism for handling compound requests.

Route ‹1,2,6,8›: The reader decides that no further requirements need to be fulfilled.

Here, in the first route all 'yes' choices are taken, in the second route all 'no' choices are taken, i.e. in the first route every possible requirement that could be fulfilled has been fulfilled, in the second route the request handling framework is left as-is.

The key difference between the two stories presented is that “Varying Design Choices” presents the reader with a fixed set of requirements, and choices which include design alternatives and sub-optimal choices for the context. “Varying Requirements” only has optimal choices, but allows the reader to choose which requirements they care about.

This distinction highlights the purpose of each story; in the first case, the aim is to encourage the reader to learn about design by making mistakes. Going down the wrong path in this story is a good thing because the reader will gain an understanding of the negative consequences of their decision. Not only that, but cheating may also be a good thing – after going down the wrong path, the reader can choose to backtrack and change their mind, exposing them to the positive consequences of decision they chose not to make. Subsequent readings of significantly different routes, such as those relating ‘horror story’ designs, may also give the reader further insight.

The aim of the second story is to provide a framework for designing actual systems; as such there are no wrong choices, and every ending is equal. Here, negative consequences are avoided in favour of presenting the many different good designs that are possible for a particular pattern language and context. However the lack of design choices in the second story is a little artificial. A real-world application of an interactive pattern story is likely to require the choice of both requirements and design alternatives; the lack of design choices in the second story in this paper serves to allow the features of interactive pattern stories to become apparent in contrast to the first story.

Benefits and Liabilities

The main benefit of the approach is considered to be the engaging format, as well as the opportunity to explore pattern language based designs.

Interactive stories in the *"Choose Your Own Adventure"* format are written in a second person, genderless way. This avoids the dry, often uninteresting tone of 'third person passive' writing. The authors of [POSA5] advise that *"A pattern description that is hard to read, passive to the point of comatose, formal, and aloof is likely to disengage the reader"* - a story written about YOU is considered to be more engaging.

The ability to make decisions in the story is also involving because the reader affects the outcome. The story takes on a game-like element where the set of outcomes is constrained by the reader's choices. In addition to being involving, this also makes the story fun to read.

The decision making mechanism also provides the opportunity to explore the various designs that are possible for a particular pattern language, as well as to experience (to a limited degree) the negative consequences of sub-optimal design choices for a given context. Although the examples presented here are relatively simple, it is feasible that more complex and thorough interactive stories could be written.

The main liability of the approach is considered to be the complexity of the task. Even writing the simple interactive stories for this paper was a non-trivial task, requiring many different possibilities to be considered and accounted for in the stories. Interactive pattern stories are likely to be difficult to modify after creation for the same reason. The complexity of the task therefore, may limit the practical applicability of such an approach. Few industrial projects are likely to invest the necessary effort to create and maintain such stories, suggesting that the approach may be better suited to academic and educational fields. That said, using the complete pattern story from [POSA5] and the pattern language in [POSA4] as a starting point did simplify the writing process considerably, and other support mechanisms such as tooling may increase the feasibility of the approach. For example the Storyspace or iWriter tools [Storyspace] [iWriter] may help to simplify interactive story development, and avoid repetition such as that found at step 14 of "Varying Requirements".

Another liability is that individual patterns or full designs from an interactive pattern story could be naively applied in an unsuitable context. For example the consequences of applying NULL OBJECT versus conditional null checking would be different if performance was a priority rather than understandability or maintainability. Such misapplication could lead to unexpected and undesirable consequences.

An interactive pattern story could also be applied in a prescriptive way to limit design options, for example to force designers to always use STRATEGY to support transparent logging policies. Such a use is likely to be unwelcome as it would be considered to be a 'strait-jacket', unnecessarily restricting design choices.

Applicability

By extension from their non-interactive counterparts, interactive pattern stories are likely to be most useful for education and learning. The ability to explore a constrained design space in a fun, engaging way suggests that interactive pattern stories will be a useful addition to teaching and learning environments.

It is expected that different audiences will benefit in different (and multiple) ways from reading interactive pattern stories, so there are potentially as many applications as target audiences. By varying the content, choices, or emphasis of interactive pattern stories, different aspects of software design and development may be targeted. The stories in this paper presented choices around design and requirements, but it would also have been possible to present choices around desired qualities. It may also be desirable to create stories combining design, requirement, and quality choices in order to more closely match real world software development.

The addition of code or model fragments and the creative use of typesetting such as italicising topic sentences may support educational applications further.

Interactive pattern stories may also serve as the basis of single narrative stories, which may be desirable for some readers. Interactive stories written with tooling support would be suitable candidates for generating such single-narrative stories, as long as the tooling supported such functionality.

Additionally, it may be possible to employ the approach to software architecture evaluation and comparison. Where patterns are applied to create a software system, a pattern story may be written to capture the design choices made. It would then be possible to introduce alternative steps to describe other potential outcomes, for example a poor design choice that was avoided or a better design choice that was missed. Such an approach may prove useful in describing architecture rationale in an engaging way.

The approach is not thought to be well suited to technical documentation because of the effort involved in creating and updating such documentation. Again, tool support may make such documentation more feasible.

Summary

This paper proposed that interactivity can be introduced into pattern stories in order to engage readers and support the exploration of pattern languages for educational purposes. The *"Choose Your Own Adventure"* game-book format was suggested as a suitable basis for introducing interactivity.

Two interactive pattern stories were told, both based on the "request handling framework" story from [POSA5]. In the first story, the reader was able to explore design alternatives in solving a fixed set of requirements, while in the second story the reader was able to choose which requirements they wished to fulfil.

The benefits of the approach were considered to be the engaging format, the ability to

explore the different designs that can be created from a pattern language, and the opportunity to experience the negative consequences associated with sub-optimal design choices. The liabilities were considered to be the complexity of the writing task, the possibility of misapplication, and the fact that prescriptive stories may be unwelcome. The approach was considered to be applicable in educational environments, and to software architecture evaluation and comparison.

Acknowledgements

Thanks to Paris Avgeriou for providing many insights and useful feedback during the shepherding of this paper for EuroPLoP 2008, and to Kevlin Henney for providing feedback on an early version of the paper. Thanks also to the authors of POSA volumes 4 and 5 and to John Wiley & Sons Ltd for granting permission to use the request handling framework story. Thank you also to Sam Clark for providing excellent feedback and guidance on the layout of the paper. Finally thanks to Oxford University's Kellogg college for providing funding for me to attend EuroPLoP 2008.

Appendix- History of Choose Your Own Adventure books

The first book in the *"Choose Your Own Adventure"* series was *"The Cave of Time"* [CYOA2] by Edward Packard, in which the reader discovers a strange cave whilst hiking. On entering and exploring the cave, the reader is transported through time, encounters many adventures, and ultimately through their choices determines which of the forty possible endings they come to. There is an approximately equal distribution of positive, negative, and neutral endings in *"The Cave of Time"*, as shown in the figure below.

The interactive fiction format has been successfully applied to gaming several times, leading to amongst others, the long running *"Fighting Fantasy"* [FF] series.

"Choose Your Own Adventure" books are not the first example of interactivity in fiction; the short-lived *"The Adventures of You"* series, also written by Edward Packard and in conjunction with R. A. Montgomery, preceded the *"Choose Your Own Adventure"* books. Earlier notable examples are *"El Jardín de senderos que se bifurcan"* (*"The Garden of Forking Paths"*) [Borges] by Jorge Luis Borges, and "*Un conte à votre façon*" by Raymond Queneau. The latter being published in "*Oulipo: A Primer of Potential Literature"* [Oulipo08], a collection of works from the "Oulipo" [Oulipo] a french group of writers and mathematicians notable for exploring "constrained writing" techniques, used to trigger ideas and inspiration [OulipoWP].

A more recent example of work by the Oulipo can be found in *"The State of Constraint"*, published as part of *"McSweeney's Quarterly Concern"*, issue 22. This includes Paul Fournel's *"Once Upon a Colony: A Tree Story, with Some Ramifications"* where the reader's decisions determine the fate of a primitive (but happy) village which encounters western civilisation [McSweeney22].

Interactive fiction has also been applied successfully in electronic format, with numerous 'text adventures' being playable by way of virtual machines, such as the 'Z' virtual computer invented in 1979 [InformZ]. The most notable and widely acclaimed text adventure stories are those published by *Infocom* [Infocom], such as *"The Hitchhiker's Guide to the Galaxy"*, developed by Steve Meretzky and Douglas Adams. An online version of *"The Hitchhiker's Guide to the Galaxy"* is available via the official Douglas Adams website [H2G2].

For further information, see *"Twisty Little Passages: An Approach to Interactive Fiction"* [Montfort] which explores interactive fiction in detail.

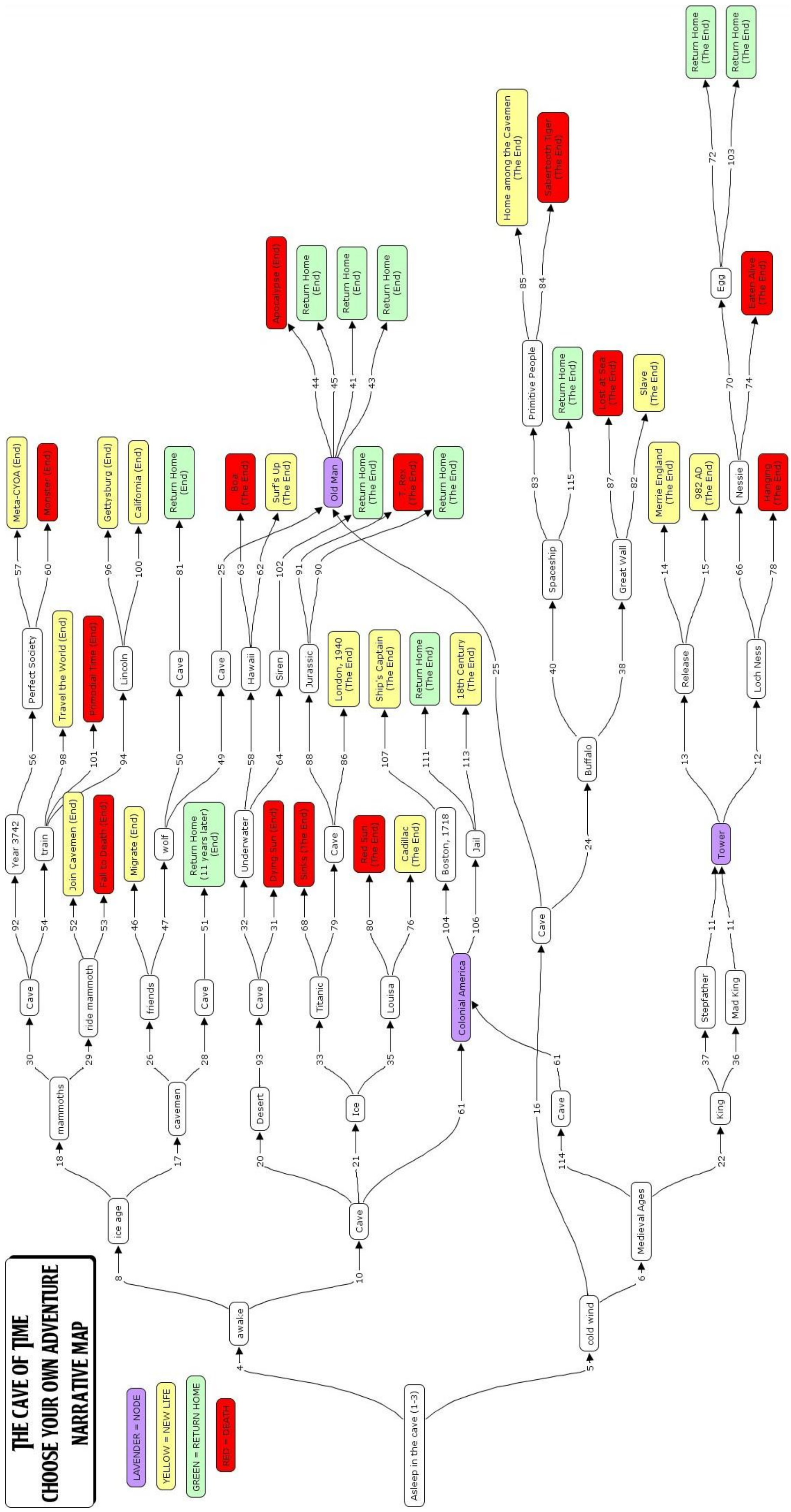


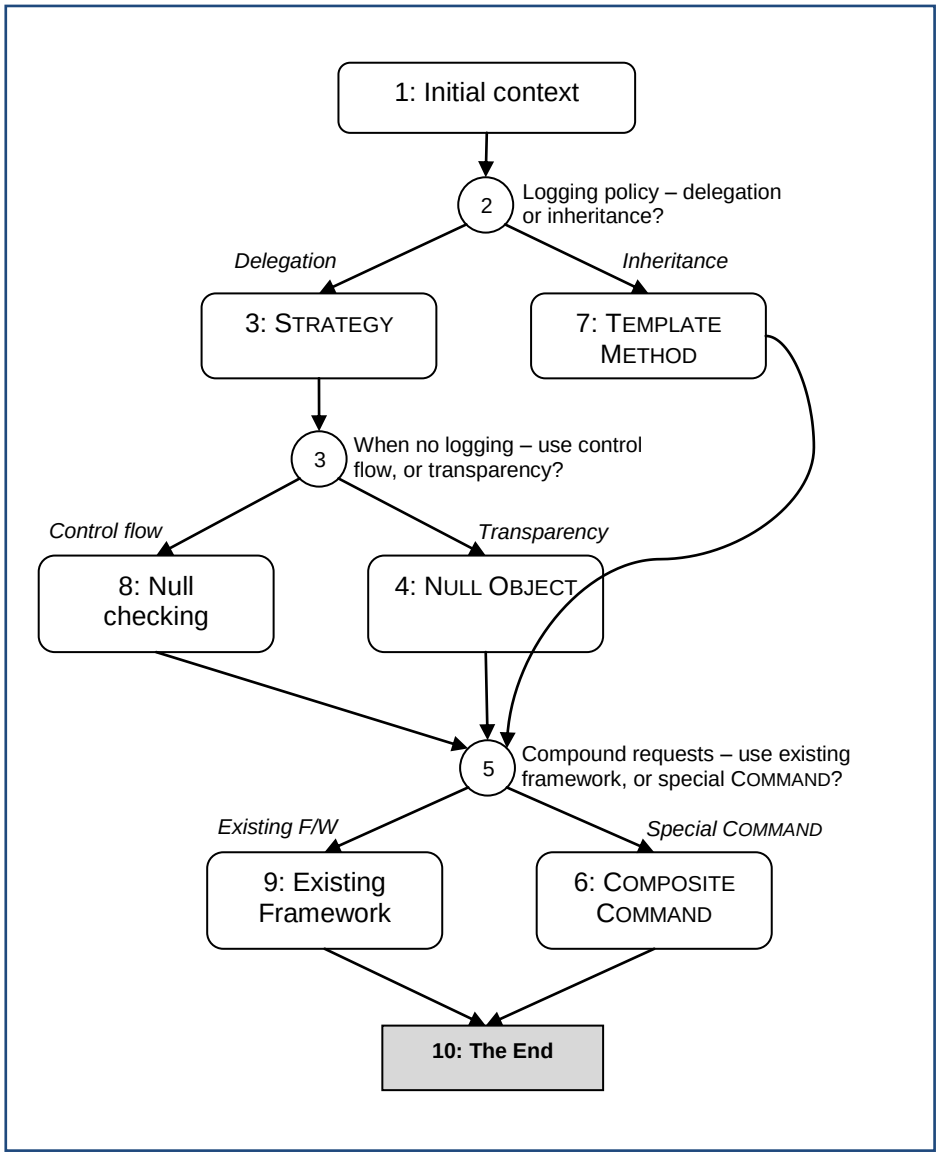
Figure 1- Map courtesy of Assistant Prof. Mark Sample from George Mason University, built using CMap [CMap].

Appendix – Story Maps

The following diagrams provide an overview of the decisions that you can make and the different routes through the stories that can be found in this paper.

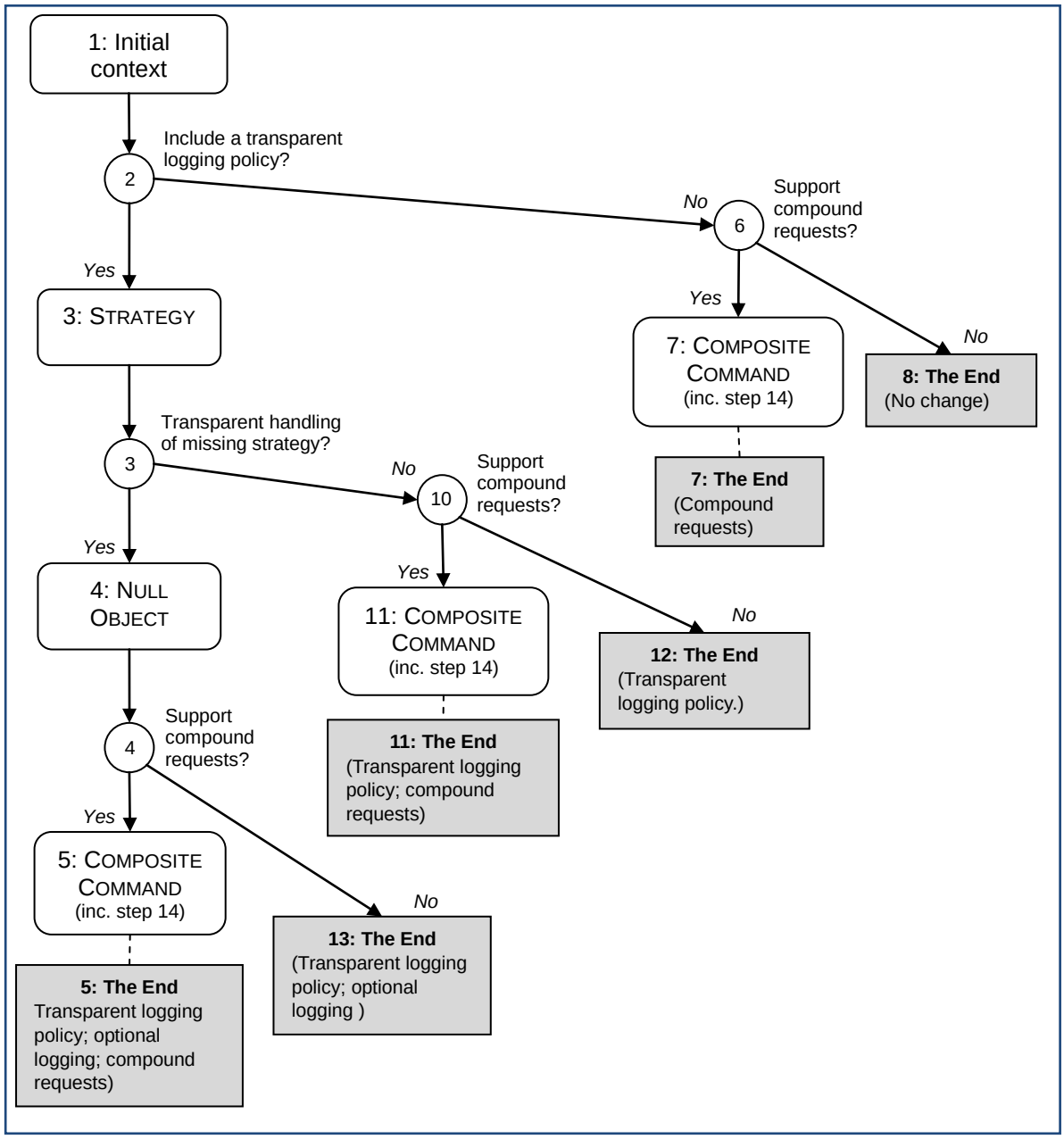
In each diagram, circles represent decisions points and italicised text shows possible choices. Rounded boxes represent the resulting development activities and decision consequences, and numbers denote discrete steps in the interactive story. Where numbered steps include both development activities and choices, the numbers are repeated. Grey boxes represent text that summarises the story at the end.

Map of "Varying Design Choices" Story



Map of Story 1 – Varying Design Choices

Map of "Varying Requirements" Story



Map of Story 2 – Varying Requirements

Note that above, step 14 (not shown) is actually referred to from several other steps.

Appendix – Pattern Thumbnails

The various patterns discussed in this paper are fully captured in [POSA4]; however for the purposes of this paper these patterns are paraphrased below:

COMMAND	When decoupling the sender of a request from its receiver, encapsulate requests being made into command objects. Provide these command objects with a common interface to execute the requests that they represent.
EXPLICIT INTERFACE	To enable component reuse, whilst avoiding unnecessary coupling to component internals, separate the declared interface of a component from its implementation
COMMAND PROCESSOR	When an application can receive requests from multiple clients, provide a command processor to execute requests on client's behalf within the constraints of the application.
COLLECTIONS FOR STATES	For objects that need to be operated on collectively with regard to their current state, represent each state of interest by a separate collection that refers to all objects in that state.
MEMENTO	To enable the recording of an object's internal state without breaking encapsulation, snapshot and encapsulate the relevant state within a separate memento object. Pass this memento to the object's clients rather than providing direct access to internal state.
STRATEGY	Where an object has a common core, but may vary in some behavioural aspects, capture the varying behavioural aspects in a set of strategy classes, plug in an appropriate instance, then delegate execution of the variant behaviour to the appropriate strategy object.
TEMPLATE METHOD	Where an object has a common core, but may vary in some behavioural aspects, create a superclass that expresses the common behavioural core then delegate execution of behavioural variants to hook methods that are overridden by subclasses.
NULL OBJECT	If some object behaviour will only execute when a particular object exists, create and use a null object instead of checking for null object references. This avoids the unnecessary introduction of complex and repetitious null checking.
COMPOSITE COMMAND	When a transparent and simple mechanism for single and compound request execution is needed, express requests as COMMANDs, and group multiple COMMANDs in a COMPOSITE to ensure that single and multiple requests are treated uniformly.

References

[Alexander+77] C. Alexander, S. Ishikawa, M. Silverstein, et al *"A Pattern Language"*, Oxford University Press, 1997

[Bass+03] L. Bass, P. Clements, R. Kazman, *"Software Architecture in Practice, 2nd Edition"*, Addison Wesley 2003

[Borges] J. L. Borges, *"El Jardín de senderos que se bifurcan"* (*"The Garden of Forking Paths"*), published in *"Ficciones"*, Grove Press / Atlantic Monthly Press (30 Jun 2000)

[BorgesHT] Hypertext version of *"The Garden of Forking Paths"* by J. L. Borges: http://www.geocities.com/papanagnou/cover.htm

[CMap] *CmapTools*, knowledge modeling toolkit: http://cmap.ihmc.us/

[CYOA1] The Official *"Choose Your Own Adventure"* website: http://www.cyoa.com/

[CYOA2] E. Packard, *"Choose Your Own Adventure 1: The Cave of Time"*, Bantam Books, 1979

[FF] Website of *"Fighting Fantasy"* gamebooks: http://www.fightingfantasygamebooks.com/

[Henney06] K. Henney, *"Context Encapsulation. Three Stories, a Language, and Some Sequences"* (2006)

[H2G2] *"The Hitchhiker's Guide to the Galaxy"* Infocom text adventure: http://www.douglasadams.com/creations/infocomjava.html

[InformZ] Website of the *Inform* system for interactive fiction: http://www.inform-fiction.org/zmachine/index.html

[Infocom] *Infocom* website: http://www.csd.uwo.ca/Infocom/

[iWriter] iWriter by talkingpanda software: http://talkingpanda.com/iwriter/

[Jackson+] S. Jackson, I. Livingstone, *"The Warlock of Firetop Mountain"*, Wizard Books; 25th Anniversary Edition (2 Aug 2007)

[McSweeney22] Various authors, *"McSweeney's Quarterly Concern"*, issue no. 22. Hamish Hamilton/Penguin Books, 2006.

[Montfort] N. Montfort, *"Twisty little passages: An Approach to Interactive Fiction"*, The MIT Press (December 1, 2003)

[Oulipo] Website of the *"Ouvroir de Littérature Potentielle"*: http://www.nous.org.uk/oulipo.html

[OulipoWP] Wikipedia entry on the Oulipo: http://en.wikipedia.org/wiki/Oulipo

[Oulipo08] W. F. Motte Jr. (editor), *"Oulipo: A Primer of Potential Literature"*, Dalkey Archive Pr; First Dalkey Archive edition (March 10, 2008)

[POSA4] F.Buschmann, K. Henney, D.C. Schmidt, *"Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing"*, John Wiley and Sons (2007)

[POSA5] F.Buschmann, K. Henney, D.C. Schmidt, *"Pattern-Oriented Software Architecture Volume 5: On Patterns and Pattern Languages"*, John Wiley and Sons (2007)

[Storyspace] Storyspace website: http://www.eastgate.com/Storyspace.html

Patterns and their impact on system concerns

Michael Weiss
Department of Systems and Computer Engineering
Carleton University, Ottawa, Canada
weiss@sce.carleton.ca

Abstract

Making the link between architectural decisions and system concerns explicit is a major contribution that patterns can make. Over the past decade, there have been several efforts to close the gap between requirements and architecture by using patterns. In this paper, our goal is to take a step back and survey these different contributions, as well as related efforts in other communities (such as the work on aspect-oriented requirements engineering). From these, we identify common elements and present a perspective on how to move forward. This thematic track on Pragmatic and Systematic Approaches in Applying Patterns should provide a good conduit for this discussion.

1 Introduction

There has been much recent interest in understanding the link between patterns and system concerns, also known as non-functional requirements. There is a well-recognized gap between requirements and architecture. We also know that system concerns may be satisfied to a differing extent by alternative architectures, and that we need to explore and evaluate architectural alternatives (Grau and Franch 2007). The system architect is faced with designing a system that meets both functional and non-functional requirements.

Harrison and Avgeriou (2007) suggest that patterns are a good way to understand the impact of architectural decisions, because they contain information about consequences and context of the pattern usage. However, they also go on to state that this information has been of limited use, because it is not presented consistently or systematically at present. They propose to integrate the information about the impact of patterns on system concerns in order to increase the usefulness of architectural patterns.

Over the past decade, a number of research groups have made contributions to our understanding of the link between patterns and system concerns. However, their work has been dispersed and we have not leveraged the results as well as we could have. As a step towards advancing these efforts, our goal is to summarize the existing research on the problem and to identify lessons learned and questions for future research.

We have divided the surveyed contributions into three streams. The first stream is on work that explicitly aims to link patterns and system concerns. Much of this work has been carried out with the goal of supporting the selection of patterns, our second stream. The third stream is concerned with work on documenting the rationale for architectural decisions and trade-offs. Here, we will only review some representative examples.

2 Patterns and system concerns

Several papers are concerned with making an explicit link between patterns and system concerns.[1] There are three perspectives within this stream: non-functional requirements modeling (Gross and Yu 2001; Araujo and Weiss 2002; Chung et al. 2002; Mussbacher, Amyot and Weiss 2006), layered system architecture and non-functional patterns (Fernandez 2003), and effective information organization (Harrison and Avgeriou 2007).

Gross and Yu (2001) examine the applicability of the well-established Non-Functional Requirements (NFR) framework by Chung et al. (2000) to the representation and application of patterns. The NFR framework makes the relationship between non-functional requirements and design decisions explicit. Gross and Yu extract the contributions of a pattern on non-functional requirements from a textual analysis of the problem statement. They then model the impact of a pattern in terms of "softgoals". Softgoal is the term used by Chung et al. (2000) to indicate that, unlike functional requirements, non-functional requirements cannot be achieved in an absolute sense, but only to some degree. Gross and Yu (2001) use softgoals to represent the forces that a pattern helps achieve or prevents from achieving. Solutions of patterns are represented as operationalizing goals. They are said to "operationalize" goals, as they turn those goals into solutions that help achieve those goals in a specific manner. Side effects of a solution are also made explicit as part of their analysis. This approach allows the comparison and consequent selection of patterns in terms of their impact on system concerns.

Araujo and Weiss (2002) improve on the work by Gross and Yu (2001) in an effort to create a catalog of the impact of patterns on system concerns using a consistent vocabulary of forces for a given domain (their domain is distributed system design). They show how patterns can be mapped to architectural issues and decisions, resources, constraints, and system concerns. Like Gross and Yu (2001), they model patterns using softgoal graphs. A link between a pattern and a force in the goal graph (which the authors call a "force hierarchy") indicates that the pattern contributes to its achievement. Each pattern is the result of a trade-off or balance between forces. Representing the contributions of a pattern as a softgoal graph makes the contributions of the pattern toward achieving the domain forces explicit. It highlights the trade-offs made by a

[1] I thank my shepherd for pointing out that these patterns are also, in some sense, about the selection of patterns. Representing the impact on system concerns is a precondition for selecting patterns. For example, when a security pattern mitigates a particular security threat, this pattern becomes a candidate to be selected when this threat is faced. However, none of the papers in this section directly discusses the application to selection. Yet, clearly pattern selection builds on pattern representations such as those developed here.

pattern. For example, it may achieve certain forces, but hinders the achievement of other forces. It also makes visible forces that remain unresolved after applying a pattern.

Chung et al. (2002) document the rationale for selecting design patterns that are used together (something they call a "pattern set") using softgoal graphs. Their approach marries goal-oriented modeling with design reuse in the form of patterns. The approach is also based on the NFR framework. It proposes to model the functional and non-functional requirements of a system using the NFR framework, refine and prioritize them, and establish architectural alternatives that meet these requirements. Next, a system designer should consider patterns that satisfy these architectural alternatives, and analyze the trade-offs among the architectural alternatives and their associated patterns. The approach ends with the selection of architectures and patterns that best satisfy the non-functional requirements identified, and instantiating the patterns in the design. For example, indirect and direct invocation are two architectural alternatives to notify subscribers, and the Observer pattern is a way of implementing the indirect invocation style. Indirect invocation leads to a loosely coupled system, which improves maintainability. This link is modeled through contributions. Pattern dependencies are also accounted for in this approach, so selecting an Observer pattern would imply using a Factory pattern.

System concerns are impacted at all levels of a system, as pointed out by Fernandez (2006). His particular focus is on security: access control and authorization constraints defined at the application level need to be enforced by lower levels, such as database, distribution, and hardware levels. Patterns provide a systematic way of reusing design knowledge to build systems that meet specific non-functional requirements. Extending the proposal of Araujo and Weiss (2002), Fernandez' approach also incorporates the notion of mapping between patterns at different levels of abstraction:

> We can define patterns at all levels. This allows a designer to make sure that all levels are secured, and also makes easier propagating down the high-level constraints.

For example, the implementation of the Authorization pattern at the application level requires the use of the Single Access Point and Check Point patterns at the system level, as well as patterns for file access and process creation at the operating system level.

Later, Mussbacher, Amyot and Weiss (2006) more clearly distinguish between a force and a non-functional requirement than earlier work. They formalize architectural patterns with the Goal-oriented Requirements Language (GRL). Forces and contributions of individual patterns are captured using GRL. Combinations and side effects (correlations) are described with AND graphs, and alternative combinations for a given (functional) goal are represented with an OR graph. With the help of strategies (that is, initial selections of candidate patterns) and propagation rules, designers can assess the impact of their selection on the forces and find a suitable solution in their context. This context can itself be modeled with GRL, first at the actor/dependency level and then at the level of intentional elements (goals, softgoals, tasks, etc.) for the system. This enables global and rigorous assessments to be made, even when many functional subgoals are considered.

Harrison and Avgeriou (2007) analyze the impact of patterns on system concerns and propose a way of organizing this information so that it is more accessible and informative. They selected well-known architectural patterns and documented the consequences of applying these patterns in terms of their strengths and liabilities in the form of tables that allow for easy comparison. Commenting on their analysis, they remark that using patterns makes it less likely that architects overlook important consequences of architectural decisions. In their words, this "relieves the architect of the burden of being expert in all the quality attributes". In comparison to other methods that center around system concerns such as QASAR (Bosch 2000), patterns focus more on the interaction among patterns and quality attributes than on specific system concerns.

Table 1 compares these approaches in terms of their features.

3 Selection of patterns

Other approaches also target the selection of patterns, and are, thus, presented in this section, although they all include a representation of the system concerns impacted by a pattern. This stream includes work on pattern-based design (Weiss 2003), design space visualization (Zdun 2006), architectural decision trees (Fernandez et al. 2006), decision-theoretic approaches to automate pattern selection (MacPhail and Deugo 2001), and pattern search engines (Weiss and Mouratidis, 2008). Note that we limited our attention to approaches that use system concerns as part of their decision process. There are other approaches to pattern selection that do not consider system concerns.

Weiss (2003) describes a pattern-based approach to system design that is both goal-driven (top-down) and pattern-driven (bottom-up) as in **Error! Reference source not found.**. Their approach involves five steps: identify domain forces, document roles (patterns are documented as role diagrams in this approach), document patterns and their dependencies, identify the overall design goals (expressed in terms of the forces implied by the requirements), and select patterns that help achieve them. The last step is concerned with selecting patterns. The first three steps are steps that only pattern writers go through, whereas the last two steps are performed by designers, who want to apply the patterns.

Having identified the overall prioritized design goals, the architect should now select the patterns that help achieve them. As in Araujo and Weiss (2002), the approach relies on a softgoal representation of the patterns. The selection is performed manually with the help of a reverse index that lists the patterns achieving a particular force. This index can be derived from the individual softgoal graph model of each pattern. Weiss (2003) also remarks that if we want to evaluate the effect of applying several patterns, we can combine the softgoal graphs for the individual patterns, and obtain a softgoal graph in which the patterns are operationalizations (designs or implementations that achieve the softgoals). We can also compare the results of applying alternative solutions to the same problem suggested by different patterns. The choice of the pattern depends on the prioritization of the forces by the designer (that is, there is no single best solution).

	Gross and Yu (2001)	Araujo and Weiss (2002)	Chung et al. (2003)	Mussbacher et al. (2006)	Zdun (2007)	Harrison and Avgeriou (2007)
Forces	Softgoals	Softgoals	Softgoals	Softgoals	Criteria	Columns
Functional goals	Goals			Goals	(Questions?)	
Patterns	Operationali-zations	Operationali-zations	Operationali-zations	Tasks	Options	Rows
Pattern dependencies			Contributions	Task decomposition, actor dependencies	Follow-up questions	
Force relationships	Contributions	Contributions	Contributions	Goal graph with stakeholders		
Context		Goal graph	Priorities	Actor dependencies		

Table 1. Features of the different pattern representations

Fernandez et al. (2006) propose the use of architectural decision trees to record selected patterns as well as alternatives that were considered but discarded. A decision tree allows architects to make decisions about system concerns vs. functional decisions. Architects can also later backtrack in the tree and make different decisions as the outcome of a decision was not the expected one or the requirements change.

Zdun (2007) describes an approach to reduce the complexity of pattern selection by employing pattern language grammars and design spaces. The approach considers quality goals (which the author equates with forces) and pattern variants. The design space approach extends the question-option-criteria (QOC) notation from HCI, which is related to the goal-question-metric approach from software engineering. Instead of using QOC analysis to visualize alternative design decisions, Zdun (2007) applies it to document the impact of alternative patterns to the quality attributes in forces and consequences. As in the work of Gross and Yu (2001) and Araujo and Weiss (2001), the level of abstraction is, therefore, that of patterns, not that of concrete design decisions. The design space approach is recursively applied, if related patterns raise new design questions.

Some proposals have been made to automate the selection of patterns. For example, McPhail and Deugo (2001) use a weighted distance metric (where each force is weighted by its priority) to search for matching patterns among a large number of patterns. An interesting aspect of their proposal is to decompose forces (such as performance and maintainability) into object-oriented quality metrics. The level of satisfaction of a force can thus be automatically computed from the object model of the pattern solution. Their approach is particularly suitable to compare variants of a pattern, that is, to determine which of various versions of, say, the Visitor pattern is best for a particular design.

Schumacher (2003) describes an expert system for the retrieval of security patterns. He proposes a representation of meta-information for security patterns, which includes the standard context, problem, solution elements as well as pattern dependencies, but also security-specific elements such as information about the threats a pattern protects against. Through a set of inference rules that encode knowledge about the pattern elements and pattern relationships, the expert system supports navigation of patterns based on pattern relationships, and detection of conflicts and comparison of alternatives. There is also some support for the qualitative comparison of patterns in terms of non-security forces.

Current work by Weiss and Mouratidis (2008) proposes a search engine for patterns that employs the pattern representation by Mussbacher, Amyot and Weiss (2006). Patterns are represented in terms of their impact on system concerns. A rules engine is used to reason about the effect of combining patterns on system concerns, and to identify trade-offs between system concerns. Its input is a set of system concerns that need to be satisfied, and its output a set of patterns that meets all requirements, if they can be satisfied, or most of them. The search engine can produce multiple pattern sets, ranked on how they satisfy the input requirements. The reasoning process also considers pattern dependencies: one important implication is that each pattern may add new requirements of its own, which then drive the selection of further patterns.

4 Rationale for architectural decisions

This stream is concerned with related work on documenting the rationale for making architectural decisions. It also looks at efforts undertaken under the umbrella of separation of concerns. There are two groups of papers reviewed here: the work by Akerman et al. (2006), Zimmermann et al. (2007) and Brito et al. (2007), which models architectural decision making in terms of reasoning about system concerns, but does not make explicit use of patterns, and work that treats patterns as reusable architectural knowledge (Zimmermann et al. 2008; Harrison and Avgeriou 2007). The former work is included here, because it has direct bearing on how we can reason about the impact of patterns on system concerns, if we treat patterns as architecture knowledge.

Akerman et al. (2006) propose an approach to software development that focuses on architectural decisions and uses an ontology to capture the architecture. The ontology has major components for capturing stakeholder concerns, architectural assets, architectural decisions, and a transformation roadmap. They present detailed models of these components, which could provide the basis for a common vocabulary for reasoning about architectural decisions. According to the authors, a pattern catalog of the type described in (Araujo and Weiss 2002) may be a start to populate an enterprise architecture ontology. Recent work by Zimmermann et al. (2007) on an Architectural Decision Knowledge Wiki applies the theoretical framework Akerman et al. (2006) and implements it in a tool. This work considers three levels of architectural decisions: concept, technology, and asset. Concepts are patterns or abstract principles.

Zimmermann et al. (2008) combines pattern languages and architectural decision models. The proposed ArchPad method facilitates the selection of patterns and provides traceability from generic patterns to project-specific adaptations of those patterns. Patterns are treated as a source of reusable architectural knowledge, whereas architectural decision models document specific design decisions and the alternatives considered. Applying a pattern means to make an architectural decision; to address the consequences of a pattern, further architectural decisions need to be made.

The impact of architectural decisions on system concerns is also heavily researched in the aspect-oriented requirements engineering community. A recent example is Brito et al. (2007), who propose to use the Analytic Hierarchy Process to resolve conflicts between system concerns. Given a set of alternatives and a set of decision criteria, the method will determine the best alternative in a rigorous manner.

Quality attributes often interact. Changes to a system that improve one set of quality attributes usually have unforeseen side effects on quality attributes elsewhere, as noted by Harrison and Avgeriou (2007). An example of the complexity of the interaction of non-functional requirements has been documented in Dyson and Longshaw (2004).

The Non-Functional Requirements (NFR) framework in Chung et al. (2000) is a goal-oriented approach for modeling interactions between NFRs, and deriving a “good” or (with respect to the user’s priorities) optimal software architecture. It introduces the

notion of a softgoal. The prefix "soft" indicates that softgoals are often subjective in nature, unlike functional (or "hard") goals. The NFR framework is used for documenting design rationale, and it helps represent the relationships between design decisions and non-functional requirements. Its extension within the Goal-oriented Requirements Language (GRL) can also model the viewpoints of multiple stakeholders (GRL 2007).

5 Lessons Learned

Our first set of lessons learned from our survey of the literature indicates that the literature on patterns and system concerns is still fragmented:

- There are several dispersed research efforts on enhancing our understanding of how to link patterns and system concerns
- These efforts lack a common vocabulary and do not agree on notation[2]
- There is also a lack of large case studies to validate the proposed approaches, specifically ones with industrial involvement

On the other hand, as this paper hopes to show, there are many common ideas underlying these approaches, and their synergy should be better exploited:

- Patterns make the communication of architectural decisions easier
- Architectural decisions are made in terms of system concerns: solutions to the same functional requirements differ in their impact on NFRs
- Patterns capture reusable architectural knowledge, so use of patterns can reduce the effort on documenting architectural decisions and help capture rationale
- There are several related notions to represent the concept of force in patterns, and there is an important distinction between force and non-functional requirement
- Pattern selection must take pattern dependencies into account (different approaches use goal decomposition and pattern language grammars)
- While forces are often treated as one-dimensional (as in "performance" is a force), they often interact in rich and complex ways
- Not all notations make the context in which a pattern is applied explicit

Acknowledgement

My thanks go to my shepherd Ed Fernandez whose probing questions and insights have helped me clarify my initial ideas.

[2] This is not to say that a variety of notations is bad, but it may be indicative of a fragmentation of the literature into different "closed" schools

References

Primary references are indicated with a (*). The other references are provided as sources supporting the argument in the paper, but are not essential reading.[3]

* Akerman, A., and Tyree, J., Using Ontology to Support Development of Software Architectures, IBM Systems Journal, 45(4), 813-825, 2006

* Araujo, I., and Weiss, M., Linking Non-Functional Requirements and Patterns, Conference on Pattern Languages of Programs (PLoP), 2002

Bass, L., Clements, P., and Kazman, R., Software Architecture in Practice, Addison Wesley, 2003

Bosch, J., Design and Use of Software Architecture- Adopting and Evolving a Product-Line Approach, Addison Wesley, 2000

Brito, I., Viera, F., Moreira, A., and Ribiero, R., Handling Conflicts in Aspectual Requirements Compositions, Transactions on Aspect-Oriented Software Design III, LNCS 4620, 144–166, 2007

Chung, L., Nixon, B., Yu, E., and Mylopoulos, J., Non-Functional Requirements in Software Engineering, Kluwer, 2000

* Chung, L., Supakkul, S., and Yu, A., Good Software Architecting: Goals, Objects, and Patterns, Information, Computing & Communication Technology Symposium, 2002

Davidsson, P., Johansson, S., and Svahnberg, M., Using the Analytic Hierarchy Process for Evaluating MAS Architecture Candidates, International Workshop on Agent Oriented Software Engineering, 2005

Dyson, P., and Longshaw, A., Architecting Enterprise Solutions, Wiley, 2004, pp. 18-22 discuss balancing non-functional requirements

* Fernandez, E., Security Patterns, International Symposium on System and Information Security, Keynote, 2006

Fernandez, E., Cholmondeley, P., and Zimmermann, O., Extending a Secure System Development Methodology to SOA, 2006

[3] In this way, I hope to balance the trade-off between the expectation that a pattern paper should only include a small number of references, and acknowledging the large number of sources that have inspired and shaped this paper.

Grau, G., and Franch, X., A Goal-Oriented Approach for the Generation and Evaluation of Alternative Architectures, European Conference on Software Architecture, LNCS 4758, Springer, 139-155, 2007

* Gross, D., and Yu, E., From Non-Functional Requirements to Design through Patterns, Requirements Engineering, 6(1), 18–36, 2001

GRL, http://www.cs.toronto.edu/km/GRL, last accessed in March 2007

* Harrison, N., and Avgeriou, P., Leveraging Architecture Patterns to Satisfy Quality Attributes, European Conference on Software Architecture, LNCS 4758, Springer, 263-270, 2007

McPhail, J.C., and Deugo, D., Deciding on a Pattern, International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems, LNCS 2070, 901–910. Springer, 2001

* Mussbacher, G., Amyot, D., and Weiss, M., Formalizing Architectural Patterns with the Goal-Oriented Requirement Language, Nordic Pattern Languages of Programs Conference (VikingPLoP), 2006

* Schumacher, M., Security Engineering with Patterns, LNCS 2754, Springer, 2003

Zimmermann, O., Gschwind, T., Küster, J., Leymann, F., and Schuster, N., Reusable Architectural Decision Models for Enterprise Application Development, International Conference on Software Architecture, LNCS 4880, 15-32, Springer, 2007

* Weiss, M., Pattern-Driven Design of Agent Systems: Approach and Case Study, International Conference on Advanced Information Systems Engineering, LNCS 2681, 711-723, Springer, 2003

Weiss, M., and Mouratidis, H., Selecting Security Patterns that Fulfill Security Requirements, International Conference on Requirements Engineering, 2008

* Zdun, U., Systematic Pattern Selection Using Pattern Language Grammars and Design Space Analysis, Software Practice and Experience, 27, 983-1016, 2007

Zimmermann, O., Zdun, U., Gschwind, T., and Leymann, F., Combining Pattern Languages and Reusable Architectural Decision Models into a Comprehensive and Comprehensible Design Method, Working IEEE/IFIP Conference on Software Architecture, 157-166, 2008

Experiences in Using Patterns to Support Process Experts in Wizard Creation

Birgit Zimmermann[1, 2], Christoph Rensing[1], Ralf Steinmetz[1]

[1] KOM Multimedia Communications Lab
Technische Universität Darmstadt
Merckstr. 25, 64283 Darmstadt, Germany
{birgit.zimmermann, christoph.rensing, ralf.steinmetz}@kom.tu-darmstadt.de

[2] SAP AG
SAP Research CEC Darmstadt
Bleichstr. 8, 64283 Darmstadt, Germany
birgit.zimmermann@sap.com

1 Overview

Users of software often complain that many software solutions only insufficiently support them in solving their problems and performing their tasks. This phenomenon occurs with all kinds of software. It can also be seen with tools that are especially developed to support users in performing specific tasks. Working on a tool supporting users in performing so called adaptation processes we found the same problem. (These processes are needed to adapt existing E-Learning material in order to make it suited for changed usage scenarios.) To offer tool support for adaptation processes we created a wizard based on a pattern based description formalism for adaptation processes. This paper presents our experiences with this approach. The next chapter gives an introduction to adaptation processes.

2 An Introduction to Adaptation Processes

Creating high-quality E-Learning material is a time and cost consuming task. Re-using existing material could reduce these costs. But often a one-to-one reuse of the existing material is not possible, as the new scenario of usage differs to a certain degree from the original usage scenario. Therefore, to achieve a high quality, it is necessary to adapt the existing material to the new usage scenario.

The processes needed to perform the adaptations are structured hierarchically: A process is performed by executing several process steps. These process steps can consist of smaller process steps or of atomic operations that cannot be split up into smaller units (see Figure 1 on the next page).

Let us have a look at one example process: The adaptation of E-Learning material to a changed (corporate) design consists of several process steps like exchanging logos and images

Proceedings of the 13th European Conference on Pattern Languages of Programs (EuroPLoP 2008), edited by Till Schümmer and Allan Kelly, ISSN 1613-0073 <issn-1613-0073.html>.

that do not fit, or changing fonts, backgrounds, colours, etc. The process step ”exchanging images” consists of several atomic operations like identifying all used images, testing for each image, if it fits to the requirements, finding images that have to be used instead of non-fitting images etc.

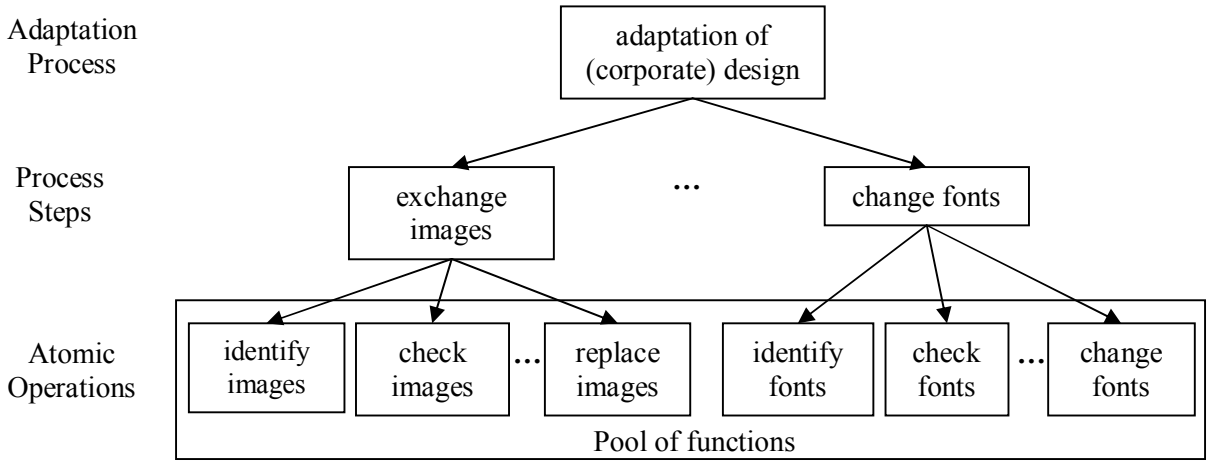


Figure 1: Process hierarchy [18].

In many cases it is necessary to perform several adaptations in order to achieve a good result. When changing the layout to make it suited for a different corporate design, it is for example often also needed to change the terminology, in order to adapt it to the terminology of the new company. But in reality, often the persons, who have to adapt the material, are not experts in performing all needed tasks. Mostly they only have a certain basic knowledge of how to perform the adaptation processes. This means that compared to Dreyfus' model of skills [4] they are novices in performing adaptation processes. But with the help of a tool that is based on the knowledge of persons from the expert level it is possible that the novices are supported in performing adaptation processes in such a way that they are enabled to achieve results that a user from a higher skill level would achieve. To develop such a tool support was our goal within the Content Sharing Project (http://www.contentsharing.com).

To be able to offer support for adaptation processes it was necessary to find out how the adaptation processes have to be performed and how a useful tool support could look like. Therefore we carried out a survey of persons being experts in performing the processes needed to adapt existing E-Learning material in order to make it suited for changed usage scenarios [18]. These persons perform adaptation processes very often. Therefore they can offer a detailed description of how to perform the processes.

Many of the adaptation process experts mentioned that existing tools are not well suited to support users in performing the adaptation processes. Especially novice users are often not able to use these tools. The main reason for this problem is that the tools often do not represent the processes themselves. They are based on the software designers understanding of the processes [9]. But in many cases this understanding differs from the processes as they are performed and understood by the process experts [11]. Thus we decided to look for a possibility that allows adaptation process experts to be involved more directly in the development of a tool offering support for adaptation processes.

3 Problem: How to Enable Process Experts to Describe their Knowledge of Adaptation Processes?

Traditional software development often starts with analysis activities in order to collect information about how processes are performed. The requirements from user side as well as from system side are collected. Based on these requirements, models of the processes are developed and implemented [12]. The exact proceeding can deviate depending on the underlying project structure (e.g. software development according to v-model, spiral models, extreme programming, rational unified process etc.).

But this proceeding often leads to problems caused by misunderstandings between process experts and software designers and developers [11]. To make things worse, it is often not possible for process experts to control if the models really describe their processes, as they do not have knowledge of common modelling formalisms like UML or ARIS. According to Siau et al. [16] UML is too complex in many cases and its constructs are ambiguous. The same holds for ARIS [13].

Someone who for example is working as a translator (an adaptation that occurs very often) is a process expert for translation. Normally this person is not simultaneously an experienced software developer or designer. Therefore most translators are not able to deal with common modelling formalisms as UML, ARIS or BPMN [1], as they do not need these methods in their daily work. The same holds for many other adaptation processes: Mostly persons have expertise in the processes they deal with in their daily work. Thus they are able to give a detailed description of how they perform these processes. But many of these persons have never learned to describe their processes with common modelling formalisms.

Thus it would be desirable to facilitate adaptation process experts to describe their process knowledge and to use this knowledge for software creation in a way that is easy to learn and easy to understand and that does not require extensive training. In order to solve this problem it was necessary to find a possibility that fulfils the following requirements:

- It must be possible for adaptation process experts to describe their knowledge with an easy to use formalism. In addition the process experts should be enabled to create a prototypic support tool based on the created descriptions. This prototype should give process experts a possibility to find out if a given process description is correctly reflected in the prototype and if a tool based on the prototype would support the described adaptation process in the desired way.

- At the same time it has to be considered that the description formalism is structured in such a way, that on one hand it is possible to create the prototype mentioned above based on the description in a way that does not require development skills. On the other hand, the prototype has to solve as a basis for further development. This means that the prototype as well as the process description have to contain all information needed to create a prototype and to allow a developer to implement a computer program by enhancing the prototype.

For the adaptation process expert it is important that the description formalism can be used without extensive training. For the developer it is important that the description formalism

allows using familiar software development proceedings and tools as far as possible. We wanted to find an approach that meets both requirements. This approach should be used to enhance the software development processes existing by now. It was not intended to replace those processes.

4 Solution: Pattern Based Process Description

Patterns document proven solutions to recurring problems; they describe best practices [8]. Patterns are noted in natural language. Hence they are easy to understand for the persons of whom knowledge is collected with the patterns [7].

Patterns offer a regular form and they can have a structured, fixed notation. They can be stored in an XML format as the Pattern Language Markup Language (PLML) [10]. PLML is an XML DTD, which originally was thought of as a common standard for HCI patterns. XML is very flexible. It offers the possibility to be used for several different areas of application. For example, XML can be rendered to HTML or other formats, e.g. by using XSLT. In this way it is readable for a non IT person. At the same time it is very structured and thus machine readable. Besides it can be imported to established UML modelling tools by using XML Metadata Interchange (XMI).

Patterns are written down in natural language. Therefore pattern based process descriptions are easy to understand for adaptation process experts. In addition software designers as well as developers will find their familiar views on pattern based process descriptions stored in XML, because of the flexibility of XML. Thus we decided to use patterns to capture the process knowledge of adaptation experts and to store these patterns in an XML format. We have written down the outcome of our user survey using patterns to describe the adaptation processes. This led to a couple of initial patterns. We revised the initial patterns together with process experts. In addition we made sure to find at least three known uses for each pattern by searching for successful applications of the patterns. Some of the results have been published at PLoP conferences ([19], [20]).

These patterns describe the processes needed to adapt existing E-Learning material on a high level. They contain important information about how to proceed. Especially they contain a section naming all needed process steps. Some other pattern formats also include sections describing concrete steps in detail. But as process steps are sometimes needed in several processes we name the steps within a pattern, but we separated their concrete description from the patterns. We added a second kind of descriptions for the process steps, called *how-to guides*. These how-to guides explain in detail what has to be done to perform each process step and which smaller process steps or atomic operations are needed during execution. Compared to the patterns describing the whole process the how-to guides are much more detailed. Most atomic operations are used in several process steps. Therefore we also have separately written down instructions for all atomic operations needed in the process steps.

Wizards are a common solution in computer science, to offer users without expert knowledge a step by step guidance through processes. According to [6] wizards can be used, if novices have to perform a complex task composed of several steps. The novices know which goals they want to reach, but they do not necessarily know which steps they have to perform. A wizard helps them in reaching their goals.

Wizards are easy to use even for users who are not familiar with the processes supported by the wizard. Adaptation processes often have to be carried out by persons who are not experts in performing these processes. Therefore they need detailed step by step guidance through the adaptation processes. Thus we decided to develop a wizard as a supporting tool for this kind of processes.

As stated before it was our aim to enable the process experts to be involved more directly into the software creation process in order to achieve software that is based on their knowledge. But we did not want them to learn common modelling formalisms or programming techniques. Thus we were searching for a possibility allowing the process experts to prove the outcome of their process descriptions. As stated before adaptation process experts often have no knowledge of common modelling formalisms. We therefore decided to develop a method that allows process experts to generate a prototypic wizard, which is based on the process descriptions. With the prototype it can be checked, whether the underlying process descriptions really describe how the process is performed. As most of the experts in performing adaptation processes do not have the knowledge to develop such a wizard, the method should be easy to use for persons without programming experience. Therefore we wanted an automated wizard generation that does not require expertise in software development. In addition the wizard should be based on the process descriptions written down in patterns, process step descriptions and atomic operations descriptions, as they contain all information needed to carry out the processes.

To be able to offer this possibility we needed a two-stage proceeding for the creation of process descriptions: First the knowledge of process experts had to be captured in an easy to understand format. In a second step this format had to be mapped to a formal XML representation allowing to generate a wizard based on the given information. Therefore it was necessary to formalize the process descriptions to such a degree that they can be used as basis for automated code generation. In addition we had to find a possibility for an easy to use automated wizard generation. In the following section we show our concept for solving this issue.

5 Implementation

The patterns, process step descriptions and atomic operations descriptions contain the knowledge of the adaptation process experts. To be able to create a wizard out of these process descriptions we needed a structured, machine readable representation of this information. Because of its flexibility we have chosen an XML notation (adapted from PLML) to store the patterns describing the adaptation processes (as described in section 4). We also used XML to store the process step descriptions and the atomic step descriptions.

But most process experts have no knowledge of XML. Because of this we developed a process description input tool (PIT), which supports users in writing down pattern like process descriptions and saving them as XML files. PIT stores the descriptions given by the process experts in two files: one file containing all textual information about the process and a second file containing a graph representation of the structure of the process. This proceeding corresponds to the two-stage proceeding for the creation of process descriptions represented before: In a first step PIT allows to capture the process knowledge in an easy to understand format. In a second step it maps the format to a formal XML representation.

The XML files are taken as input for a wizard generation tool (WGT). WGT generates a wizard by interpreting the data provided by PIT. The generated wizard represents a first prototype of a process support wizard. It can be taken as a starting point for further development.

Thus the pattern-based wizard generation approach proposed in this paper is based on a three-step proceeding:

1. Writing pattern like process descriptions
2. Automated wizard generation
3. Extension of prototypic wizard

Each of these steps is explained in detail in the next sections.

6 Step 1: Writing Process Descriptions

In the first step the process expert has to create a pattern like process description. For each process one pattern is created. If a process is more complicated and contains several sub-processes, it is possible to create patterns for the sub-processes and to link them to the super process. (This is realized via a specific type of related patterns.) In addition the process step descriptions and the descriptions of the atomic operations have to be created.

To support process experts in creating process descriptions in the needed format the process description input tool (PIT) has been created. PIT is a Java application developed under Eclipse. It offers an input form helping process experts to describe processes in the required pattern based notation formalism.

Figure 9 in the appendix presents a part of PIT's input form for process descriptions. One can see that the process description contains common pattern elements like intent, context, or problem statement. These elements provide a reason why a process can be applied, why a certain context has to be fulfilled to be able to perform the process etc. Thus they are useful for other persons who might want to perform the described process. All this information is very valuable to all persons, who have to decide, which process helps them in solving a specific problem, and who are not process experts.

As shown in figure 9, mandatory input fields are marked with an asterisk. We have chosen these elements as mandatory as they are most important to other persons in order to understand why and how a process has to be performed. Mandatory elements are:

- The **pattern ID** is a unique ID used to address the pattern. It is created automatically.
- The **name** provides a first, rough idea what the pattern is about.
- The **problem** section describes the situation addressed by the pattern.
- The **solution** explains to the user, how the problem can be solved.
- The **process steps** list all steps needed to execute the solution.
- The **consequences** help other persons, to decide if they want to apply a pattern or not, depending on if the positive consequences are more important than the negative ones.

The non mandatory elements are needed if a process expert wants to write a pattern, but they are not necessary, to offer a useful process description:

- The process expert has a certain **confidence** in the process description. (As we talk about process experts the confidence should be high.)
- The **intent** gives a short overview what a pattern is about.
- The **context** describes the situation where the pattern can occur.
- The process expert can give an **example** of a successful application of the pattern.
- The **forces** describe the sometimes contradicting trade-offs that must be considered when performing the process.
- **Known uses** describe situations where the pattern has been applied successfully.
- **Related patterns** can exist, but it might be that there are no related patterns.

When the process expert stores a process description, the given information is stored in XML files. The XML files contain in one file the textual description of the process and in a separate file the graph representation defining the process flow and all its dependencies and preconditions. In the appendix a DTD of the XML file containing the process information can be found. Many of the elements of this DTD are taken from PLML [10]. The PLML v1.1 DTD contains some additional elements that have not been taken into account here: `alias`, `synopsis`, `diagram`, `rationale`, `literature`, `pattern-link` and `management`. The patterns taken as a starting point for this work do not need these elements.

But our patterns contain some elements that have a slightly different meaning compared to PLML. The PLML element `illustration` is called `example_illustration`, as to our understanding, this better explains what is meant by this element. To be able to better differentiate the `example` as used in PLML from the `example_illustration` we call it `example_explanation`. What is referenced as an `implementation` in PLML is called `process_steps` in our DTD. The `related_patterns` already contain a link to the pattern they are related to. Therefore our `related_patterns` are a kind of a combination between `related-patterns` and `pattern-link` in PLML.

The following elements that are used in our DTD are not mentioned in PLML, but they are necessary in our approach:

- `intent`: The intent explains what the pattern aims for.
- `known_uses`: The known uses are part of the example-section in PLML. In our patterns the example only contains one known-use. Other known uses can be added by using this element.
- `consequences`: The consequences occur when the pattern has been applied. Positive and negative consequences can occur. As for a reader of a pattern it is very important to know, what will happen when applying the pattern, we added this section.

6.1 Defining Process Steps and Atomic Operations

The steps of a process, their interdependencies, and the preconditions for the execution of each step constitute the process. Thus they are essential for the execution of a process. Also branching and cycles within the process flow are of importance. Hence PIT provides a special

wizard to define the process steps. This wizard allows a fine granular specification of the process flow without requiring special knowledge of process modelling. It is started by pressing a button to add a step to the process description.

With this wizard users can define for each step if the execution of a step is mandatory or optional, or if certain preconditions have to be fulfilled, before the step can be executed, or if there are dependencies on other steps. Branches are embodied by a special kind of precondition: If the precondition leads to one result, one step is executed, if it leads to another result, another step is executed. Cycles are defined by specifying a step that is the starting point for the cycle and another one, which is the end point. In addition a termination condition has to be defined. This is again modelled by a special kind of a precondition. Figure 2 shows the screen used to define a step that has to be performed, if other steps have been performed before. Another part of the wizard helps users to define preconditions and a third one allows to model cycles. In addition an input form exists, used to describe how a process step has to be performed. The description of process steps is also stored in the process description file.

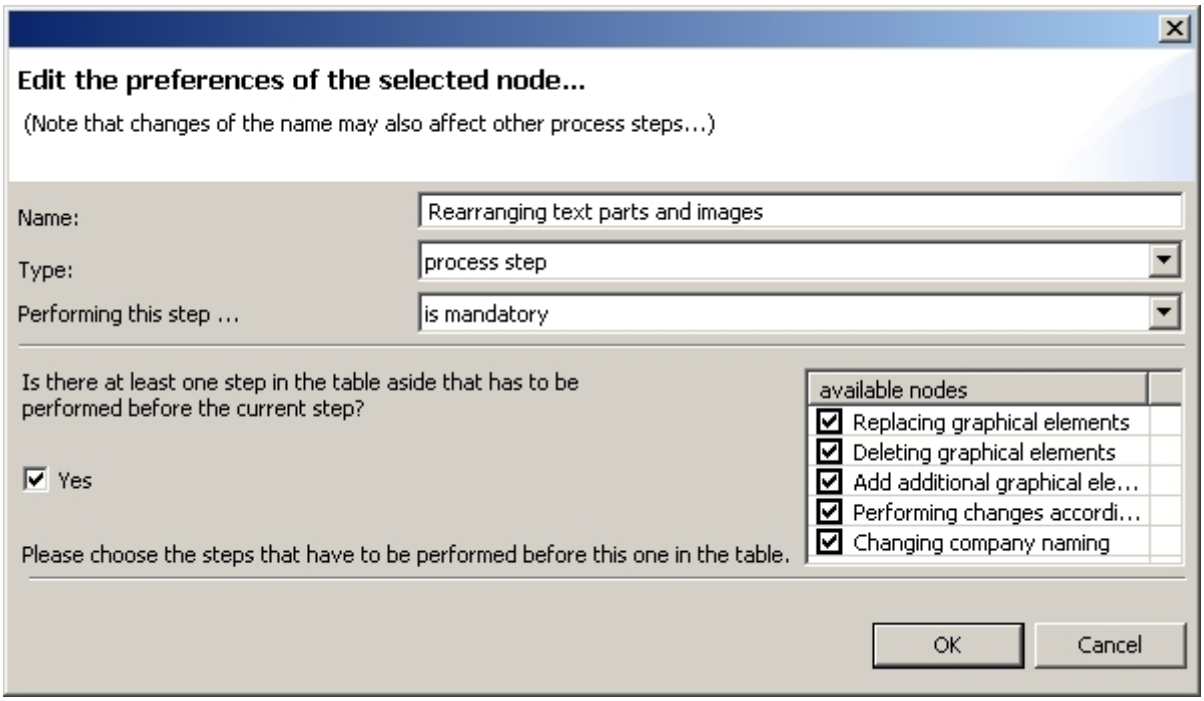


Figure 2: Defining dependencies between steps.

Compared to the pattern based description of a whole process, the process step description is very short: It contains the name of the process step, a detailed description of how to perform the process step and a listing of all smaller process steps and atomic operations needed when performing the process step. Sometimes a process step is as complex that it can be regarded as a complete process on its own. Then it is possible to create a process description for this step. This description can be added to the process step description via a special link. In the wizard this is presented as a link to an additional page containing the complex description. Users of the wizard can then read this additional information.

As steps themselves consist of smaller steps - atomic operations or again process steps - it is also possible to define all operations and smaller steps needed to perform a process step.

Again a wizard helps to state if the execution is mandatory or optional, or if certain preconditions have to be fulfilled. Dependencies on other atomic operations can also be determined. For each atomic operation a description has to be provided, how the operation is executed. This can be done via a special input form.

There exist three different kinds of atomic operations: queries, decisions and executions.

- **Queries** are needed to determine information. For example: Find all images used in an E-Learning course.
- **Decisions** are needed if a person, who performs the process, has to decide on something. For example: Decide for each image, if the image has to be deleted or not.
- **Executions** are needed whenever something is changed or done. For example: Delete all images that have been chosen for deletion.

Based on the information represented via the process steps and the atomic operations the second XML file is created. This file represents the process flow. It names all steps and atomic operations. For each step or atomic operation it contains the information whether the execution is mandatory or not, and if there exist preconditions or dependencies. Via the pattern ID, process step ID, and atomic operation ID it is possible to map the information stored in this file to the information stored in the description file. Listing 2 in the appendix contains an example for such a process graph description.

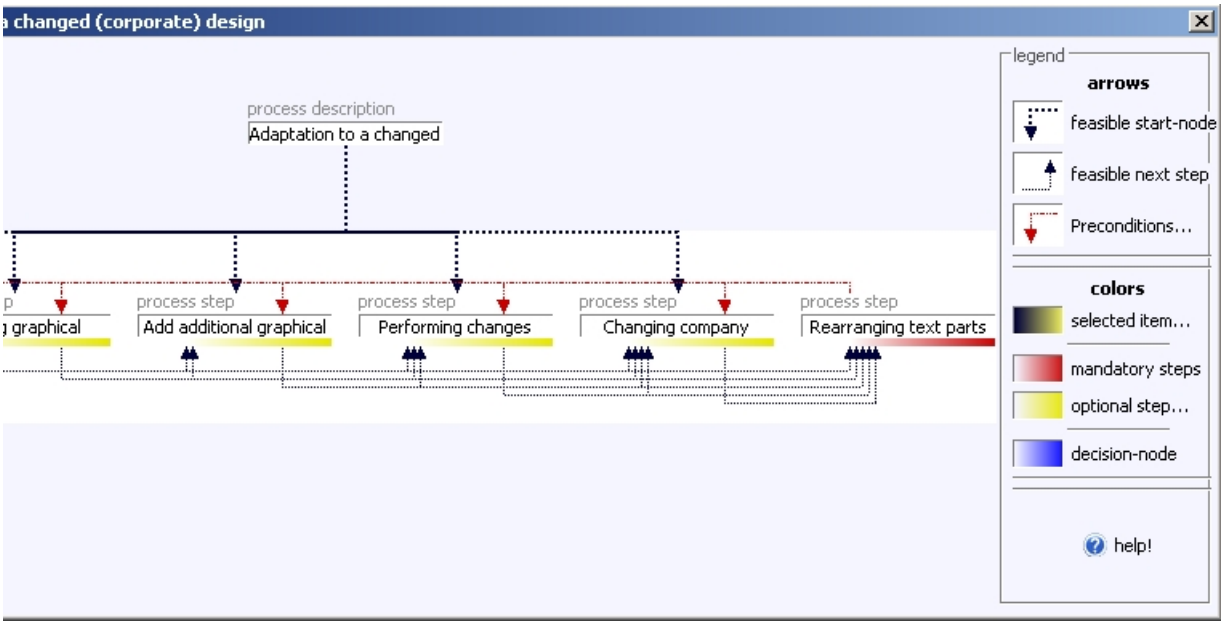


Figure 3: Part of the visualization of a process graph.

PIT offers several possibilities to work with the generated process descriptions: It is certainly possible to edit an existing description. In addition the textual description stored in XML can be rendered as HTML in order to view it via a browser. Thereby it is possible to read the entire process description as a continuous text. Thus it is more comfortable to control whether the description is complete and accurate. Furthermore, a visualization of the process graph is available that represents all steps of the process flow. (Figure 3 shows the first level of the process graph corresponding to listing 2 in the appendix. By clicking on a process step the

level under this step opens in the viewer.) At the moment this visualization is only a first prototype. Further enhancements are planned. As the process descriptions are stored in XML it is possible to offer several visualizations that are tailored to the needs of several persons (a developer and a process expert need, for example, different visualizations). In addition it is planned to offer an export to XMI. The XMI files could be used to provide a class diagram and an activity diagram representing the process.

7 Step 2: Automated Wizard Generation

The XML files generated by PIT serve as input for the wizard generation tool (WGT) used in step 2 of the proceeding proposed here. WGT is a Java application developed under Eclipse. Based on the information of PIT's XML files it generates a wizard that, as a first prototype, can be used as a starting point for further development.

WGT is started by selecting a menu entry in PIT. WGT reads the process description that is actually open within PIT. The user specifies where the generated wizard has to be stored and then starts the wizard generation by simply pressing a button (compare figure 4). WGT then parses the XML files created by PIT. It extracts the information contained in the files and fills several predefined code templates with this information. By this procedure all needed Java classes for the prototype are created. The "Activate additional options..." section shown in figure 4 can be activated to open a dialog that allows to specify, how the atomic operations have to be distributed over the wizard pages. (Later in this chapter this is explained more detailed.)

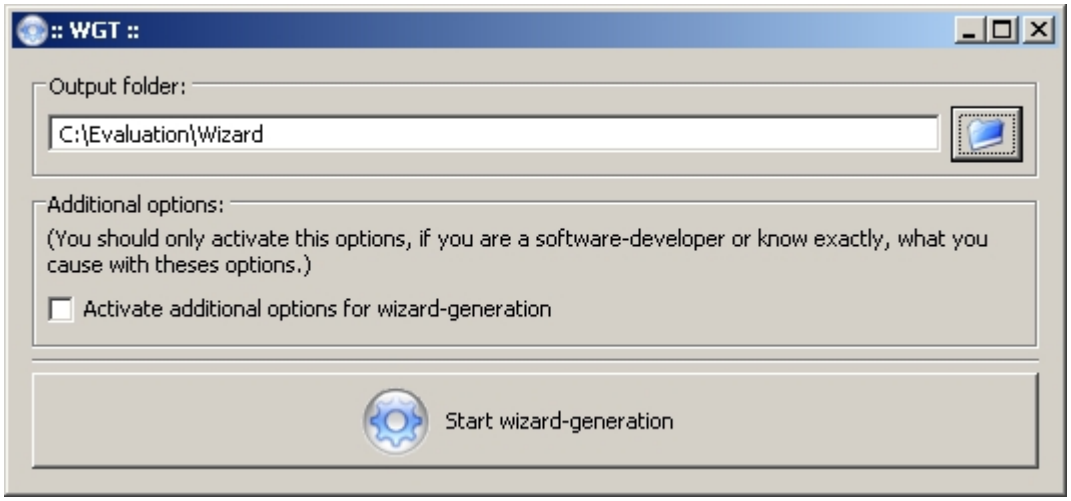


Figure 4: Screenshot of WGT.

The main purpose of the wizard generation tool is to generate a wizard based on the information given by the process descriptions created with PIT. The wizard should have a graphical user interface, which allows directly after its generation that the process expert evaluates whether the wizard contains all needed information and whether the process flow is correct. The design of the user interface should follow common rules for designing user interfaces, as described in [3], [14].

The atomic operations used in the wizard are not automated directly after generating the wizard. Instead the wizard contains descriptions telling a user how to proceed. We believe that it leads to better results, if an experienced software developer checks, where it is reasonably possible to automate certain operations. Where this is possible the developer can extend the automatically generated source code. To make it easier for the developer, to find where the source code can be extended, comments are added to the automatically generated source code during its creation.

Another important point during wizard generation is the time needed to generate the wizard. As long as the wizard does not fulfil all needs the process expert will change the description created with PIT and generate a new version of the wizard. Thus it is highly probable that the process expert will use WGT several times until a wizard is generated that really fulfils all needs. Therefore WGT has to be fast in order to reduce the time, where the process expert has to wait for the generation to finish.

WGT takes the process description as a model of the process that is supported by the wizard. This kind of software creation is called generative programming [2], which is a special kind of model driven software development (MDSD). There are a lot of different approaches for MDSD [17]. As large parts of the wizard stay the same for all kinds of processes regarded here, we decided to use code generation templates to create the wizard. This allows a fast source code generation.

The wizard generation tool consists of three logical units that are passed one after the other (shown in figure 5):

1. In the first part the XML files provided by PIT are read. The information about the process flow is transferred into an internal process structure model and the information contained in the descriptions is extracted.
2. The second part uses this information to map it to Java code templates. The textual information is used to enhance the content of the graphical user interface. The structural information is used to create all process steps and to transfer the structure of the process flow to the wizard. By this all needed Java classes are generated.
3. The third part writes the generated source code to several Java classes and marks all parts in the source code that can be enhanced by adding additional source code. In addition the created classes are compiled and a batch file is created that allows starting the wizard comfortably.

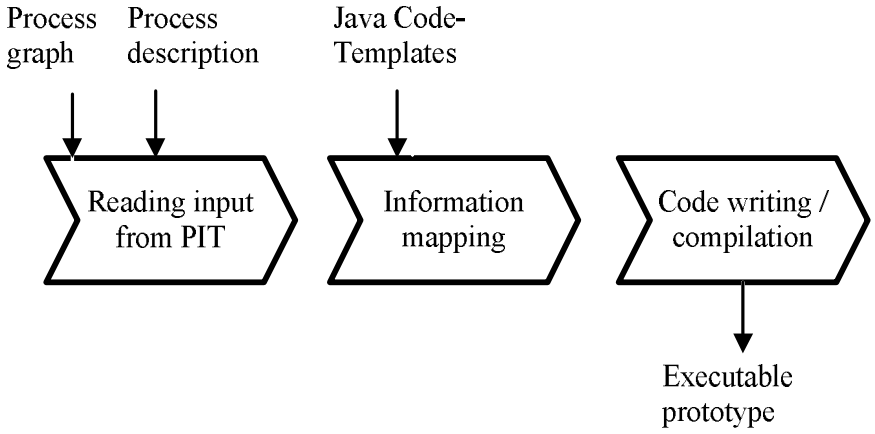


Figure 5: Three phases of wizard generation.

The prototype wizard generated by WGT is based on the model-view-controller principle [5]:

- The process graph containing information about the process flow serves as a *model*.

- The wizard pages providing a graphical user interface build the *view*. Those pages contain the textual information of the pattern-based process description. They are based on Java code templates. WGT instantiates those templates by filling them with the given information. The pages then contain a detailed description on how to perform the process, each process step, and each atomic operation.

- The *controller* of the wizard passes events caused by the user to the model and reactions caused by the model back to the user. The controller interprets the process flow information provided by the model. Depending on the user input the controller monitors which step has to be performed at which time and which step is possible as next step. Thus the controller assures a correct process flow. In addition it contains information about the actual state of the wizard, as it stores, which steps have been carried out so far as well as their results. This is important for a further implementation of the wizard enhancing it with automated functionality. The controller also has to collect and distribute all data that have been created or are required when performing several steps in an automated way.

As large parts of the source code stay the same, code generation templates exist for most wizard classes. There are several kinds of page templates and composite templates depending on the function of each part of the wizard: One template exists for the start page of the wizard, another one for the last page. A special page template exists for process steps. This template contains a part, where the description of the process step can be added. The atomic operations are realized via composites that are based on composite templates. For the three different types of atomic operations three different templates are used. The composites of the operations of a process step are grouped together on one page, which is also created based on a template.

After generating all the Java classes of the wizard, WGT starts to compile those files and to build an executable wizard. A batch file is created that allows to start the wizard comfortably. The executable wizard is a first prototype. It offers the process expert the possibility to prove, whether the wizard represents and supports the described process. If this is not the case, the process expert can refine the process description and generate a revised version of the wizard.

7.1 Arranging Steps and Operations on Pages

One problem when creating the wizard is how to distribute the process steps and atomic operations over the wizard pages. There are several possibilities how to solve this problem: One could create one page explaining each process step and one additional page for each atomic operation. As many adaptations consist of quite a lot of process steps and atomic operations, this can easily lead to a huge number of pages with sometimes only a short explanation on it. Therefore this possibility has been discarded.

Another possibility would be to create one page for each process step, and to add all atomic operations needed in the process step to this page. But many process steps contain ten and

more atomic operations. Then the pages would become huge and overcrowded with information. We also discarded this possibility.

We decided to use something in between these two extremes: We create one page describing how to perform a process step. As the wizard offers an expert and a novice mode, this page can be displayed to novice users and it can be hidden for experts, who do not need this information. In addition the atomic operations are spread over several pages. By default 5 atomic operations are grouped on one page. But someone who knows how the user interfaces of the automated atomic operations look like can group the operations in such a way that the automated operations fit well on the page. Therefore the "Additional Options" mode of WGT exists. This mode allows to group the atomic operations on the pages in a way that best fits the needs.

Figure 6 shows how a user can group atomic operations to pages. You can see that for each process step it is shown, which operations are needed in this step. On the right site it is possible to see which operations are placed on which page. By selecting an operation and pressing the arrows on the right you can move this operation to a page before or behind the actual page. Selecting for example the operation "If the size does not fit the requirements, change it" and pressing the "Down"-Arrow on the right would move this operation to the second page.

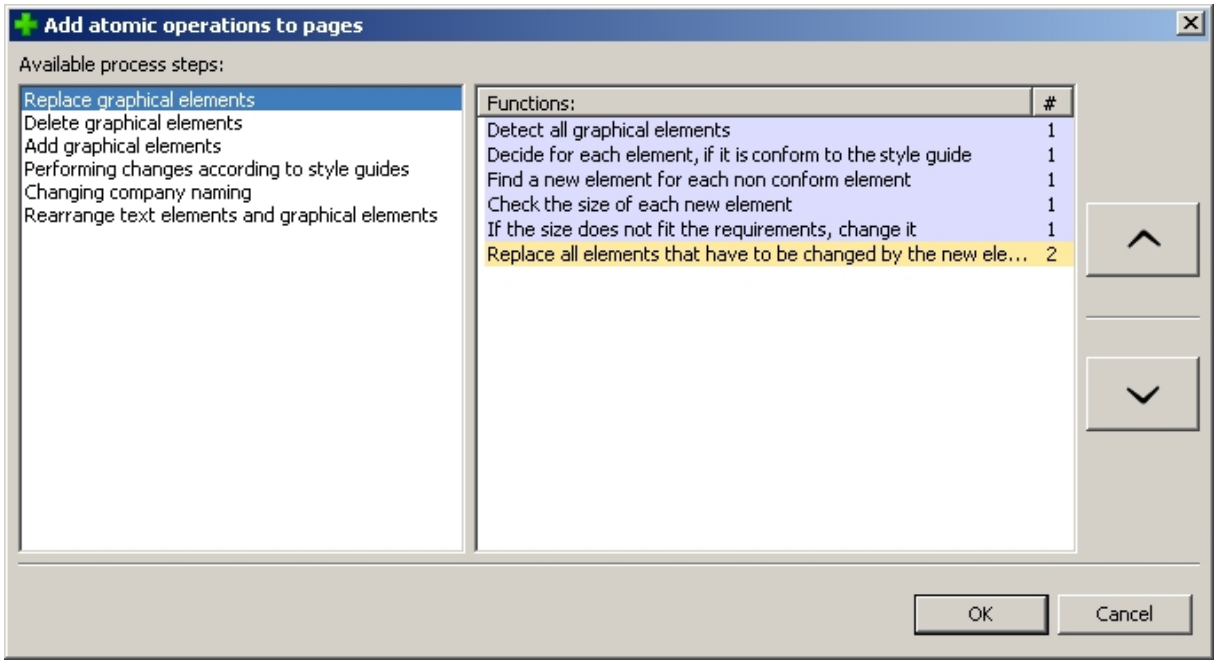


Figure 6: Arranging atomic operations on pages.

8 Step 3: Extension of Prototypic Wizard

Together with the process descriptions generated during the first step the wizard serves as a basis for communication between process experts and developers. Based on this prototype a common understanding of the process described in the wizard can be established. The wizard makes it easier to discuss ideas for further development in a vivid way. In addition it provides

a code skeleton that can be enhanced comfortably by a developer. For this purpose the code generated by WGT is marked with special comments indicating, where additional code can be added. This can be done in the third step of the proceeding presented in this paper. The two steps presented so far are both executed by process experts. Step 3 has to be executed by a developer.

To make work faster and less error prone it is useful to add automated functionalities to the wizard, where this is reasonably possible. Therefore the wizard created by a process expert has to be handed over to a developer. The developer gets the information created in the first step and the prototype generated in the second step.

With the Wizard Generation Tool WGT for each process step one or more pages have been created. The pages contain a list of all operations needed to execute the process steps. The developer can add additional source code for each operation that can be automated. The initial source code contains comments indicating where automation is possible. The developer has to provide a new part of source code describing the user interface and a code section in the controller class that stores the information needed for and provided by the operation. The operations itself are stored in a so called function pool. This allows to reuse operations in several process steps. Comments within the original source code give hints, where to enter the new code and which dependencies between model, view, and controller have to be taken into account. By this it is possible to add code for all operations that can be automated. Figure 7 and figure 8 show a part of one wizard before and after adding automation to the operations. Together with the process expert the developer can check for each operation if it works in the desired way. By this proceeding we enhanced the prototype created based on the adaptations patterns to a fully functioning support tool for adaptation processes.

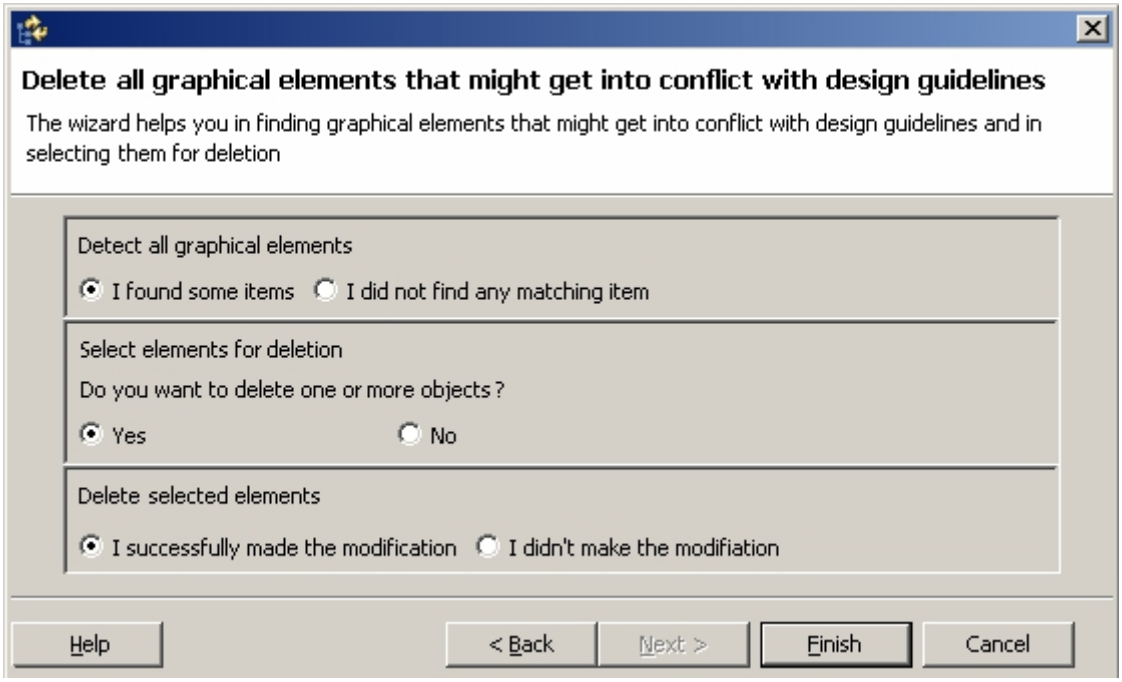


Figure 7: One wizard page before adding automated functionalities.

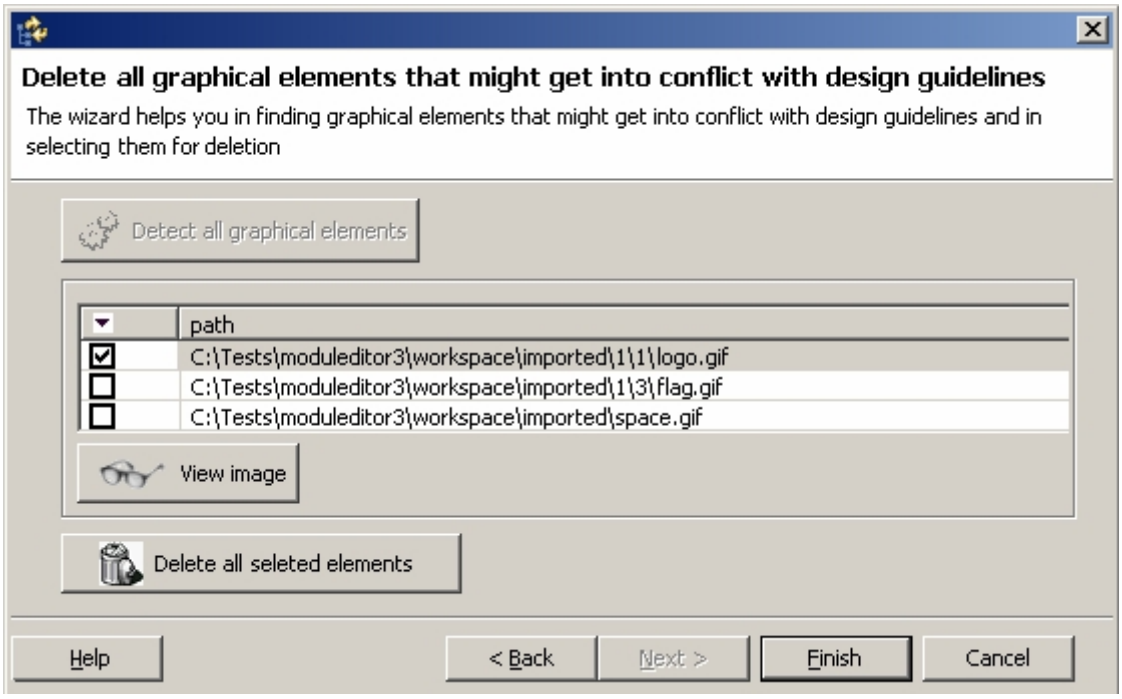


Figure 8: The same page after adding automated functionalities.

9 Application of the Approach to the Example Scenario

With the approach presented before we created a wizard supporting users in performing the adaptation processes described in chapter 2. The wizard is based on patterns describing the following adaptations:

- Adaptation to a changed (corporate) design
- Adaptation in order to achieve a print version
- Adaptation to a changed terminology
- Adaptation to a changed language (i.e. translation)
- Adaptation to achieve an accessible version

For each adaptation a pattern exists describing the adaptation process. (The adaptation to achieve an accessible version is an exception: This adaptation is described by several patterns, used to describe sub-processes.) By entering the patterns into PIT and by adding all needed additional information about the processes, like process step descriptions and atomic operations descriptions, we created process descriptions. Then we generated a wizard with WGT based on these process descriptions. Outcome was a first prototype wizard (Figure 7 shows one page of this wizard).

As a next step we automated all parts of the wizard, where this was reasonably possible. (Figure 8 shows one of the pages enriched with automated functionalities.) We analysed the functions needed to perform the supported adaptation processes and we designed functionalities that could be used to enhance the prototype. These functionalities have been added to the wizard. Outcome of this proceeding was a tool that supports users in performing adaptation processes for E-Learning material.

A first evaluation with test users of the Content Sharing Project was promising: The users were enabled to perform all offered adaptations correctly. They found process guidance and detailed help on all processes. Based on their feedback we improved the functionalities. In a second, larger evaluation we tested our tool by comparing it to a common WYSIWYG HTML editor. At the moment no tool exists that supports the adaptation of existing E-Learning material to changed usage scenarios. Therefore we have chosen a tool that supports at least many functions needed to perform the adaptation processes to compare it with our tool. As many E-Learning courses are stored in HTML format we have chosen an HTML editor. We wanted a tool that is easy to use and that offers a WYSIWYG function allowing to control directly what has been changed. We decided to use Netscape Composer as HTML editor, as it is easy to use and allows to perform at least some typical adaptations.

We asked 32 users to perform some typical adaptations to three existing E-Learning courses. (It was possible to perform all adaptations with our tool as well as with Netscape Composer.) One E-Learning course was dealing with medical topics, one was an introduction to Multimedia, and the third one was a course to learn English. Half of the users were asked to work with our tool; the others got the WYSIWYG tool. Both groups got a detailed explanation how to use their tool. The tasks were the same for both groups. At the end of the test the participants were asked to answer a questionnaire in order to determine how satisfied they were in working with the tool.

Both groups were able to perform the adaptations as described in the manuals. Both groups did the tasks fast and with only very few errors. But we found that users working with our adaptation support tool needed less time to perform the tasks (in average 14 minutes with our tool compared to in average 20 minutes with Netscape Composer). And they made fewer mistakes. (Users working with the Netscape Composer made in average twice as many mistakes as users working with the adaptation tool.) In addition they were more satisfied with the use of our tool. As the adaptation tool got very positive feedback and the outcome of the adaptations had a very good quality, we think that our tool offers a better support for adaptation processes then the tool used for comparison. In addition we think that the knowledge collected with patterns is a good support for users in performing adaptations as the explanations how to use the tools for both test groups were based on the patterns. And the results for both test groups were very satisfying. For the future we plan to enhance our adaptation tool based on the feedback we got from the users.

All users, that have been taking part in the evaluation, have knowledge of the HTML file format. But one additional benefit of our tool is that this knowledge is not needed, as the tool abstracts from a concrete file format. Thus it is possible to use one tool to perform all adaptations in all files belonging to a learning resource without having detailed knowledge how of the formats. This is a feature that is not supported by the tools that up to now have been used to perform adaptation processes.

10 Evaluation of Concept

The aim of the concept presented in this paper was, to enable adaptation process experts without knowledge of process modelling to describe adaptation processes by an easy to

understand process description formalism and to create prototypic wizards that reflect the processes as they are performed by the process experts.

To evaluate if process experts are able to use the process description formalism and the wizard generation tool independent from their knowledge of common programming formalisms and modelling formalisms, we performed a user test with 32 users. Half of the persons had knowledge of process modelling and of IT related issues like programming. Half of the users did not have this kind of knowledge.

All persons were asked to describe the same process with the help of PIT and to generate a prototype wizard based on the process descriptions by using WGT. It turned out that all users made very good process descriptions with only very few errors. Users without modelling and IT knowledge made a few more errors then users with this kind of knowledge. (In average the difference was only one error.) All users were able to generate a good prototype. It was no difference between the two user groups. In addition all users gave a very positive feedback regarding the understandability and the manageability of both tools. Thus we assume that all users were able to use the process description formalism and the wizard generation tool in the intended way. We also got some feedback how to increase the usefulness of the tools and the resulting wizard. We plan to analyse this feedback and to take it as a basis for further improvements.

11 Unresolved Issues and Future Work

There are several unresolved issues concerning PIT and WGT. For the future we plan to work on these issues. In this section we give an overview on the unresolved issues.

PIT allows entering relationships between processes. These are taken into account when generating the wizard: If one process has been finished in the wizard, a hint to related processes is given. PIT also allows specifying forces and consequences. But at the moment these are not taken into account in the generated wizard. But we are thinking on how to realize this. One possibility would be to use the forces as well as the consequences to find out if a process is useful in a specific situation:

If users are not sure which process they have to perform, they can search the problem statements. This helps to limit the number of possible processes. In addition users should be able to browse through the consequences and forces. This also reduces the number of possible processes. If a process has been performed the consequences can again be shown to users to allow them, to decide if a second process is necessary to eliminate negative consequences.

At the moment the wizard generation tool WGT is a first prototype. The layout of the wizard has to be overdone by taking into account common HCI guidelines. Nevertheless the wizard as it is by now already offers a good starting point for further development and a valuable basis for communication between process experts and software developers, as the evaluation of the adaptation tool, which is based on such a wizard, has shown.

Additionally to serving a basis for communication, the prototype wizard also can be used as a starting point for further development. Therefore WGT adds comments to the automatically generated source code of the prototype. Those comments offer hints to a developer, where it is

possible to change the source code of the prototype in order to add additional functionalities. But it might occur that a process expert decides to change the process description after a while and to create a new prototype wizard based on the changed description. The new wizard then does not contain the enhancements added by the developer before. At the moment this means that the developer again has to add the additional functionalities. For the future it is planned to develop a concept that allows to merge both versions.

We created the approach presented in this paper to support adaptation processes. But we believe that it also can be used for other kinds of processes. Thus we tested for several other kinds of processes, if it is possible to describe them with PIT.

We found that PIT also can be used to enter already existing patterns (that might have to be adapted to the pattern notation used here) as well as new patterns. We have successfully tested this with some of the security patterns presented in [15]. In addition we created new patterns describing processes for hiring new employees as well as processes for booking journeys. For all these kinds of processes it turned out that it was possible to describe them with PIT and to create prototype wizards with WGT.

But it seems that there are other processes that cannot be described with PIT, e.g. processes with a focus on data flow. Thus, for the future it would be desirable to analyse which kinds of processes can be described with PIT and for which kinds of processes the approach presented here does not work.

When we were creating the E-Learning material adaptation wizard (compare section 9), we entered patterns as descriptions for the process and additional descriptions (not patterns) for the needed process steps and atomic operations. But as PIT only reassures that all needed information to perform a certain process is provided in the predefined structure, it has to be taken into account that the process descriptions provided by process experts might not meet common pattern criteria like being generic or having at least three known uses. It might occur that the process experts only create process descriptions that are written down in a pattern based notation formalism but do not meet common pattern criteria like having at least three known uses. But these descriptions also contain valuable knowledge and offer a good basis for the wizard creation. As the process expert is enabled to generate the wizard he or she can change the process description as many times as needed to achieve a wizard that really meets the experiences of the process experts.

12 Acknowledgements

The authors thank all persons, who supported this work. Especially we thank Michael Weiss, who provided a huge amount of helpful comments during shepherding for EuroPLoP 2008. Although many thanks go to our writer's workshop group at EuroPLoP 2008 for a very good discussion and so many useful hints.

The authors thank SAP AG - SAP Research CEC Darmstadt, as well as KOM Multimedia Communications Lab at the Technical University of Darmstadt for supporting this work.

This work is supported by the German Federal Ministry of Economics and Technology in the context of the project Content Sharing.

13 References

[1] Business Process Modeling Notation Specification. Final Adopted Specification, 2006. Available under: http://www.omg.org/docs/dtc/06-02-01.pdf

[2] Czarnecki, K., Eisenecker, U. W.: Generative Programming - Methods, Tools, and Applications. Addison-Wesley, 2000

[3] DIN En ISO 9241: Ergonomics of Human System Interaction. Part 11: Guidance on usability and part 110: Dialogue principles.

[4] Dreyfus, S.E.,Dreyfus, H.L: A five−stage model of the mental activities involved in directed skill acquisition. Unpublished report supported by the Air Force Office of Scientific Research (AFSC), USAF (Contract F49620−79−C−0063), University of Califonia at Berkley, 1980.

[5] Eckstein, R.: Java SE Application Design With MVC. 2007. http://java.sun.com/developer/technicalArticles/javase/mvc/index.html

[6] Folmer, E.,van Welie M., Bosch, J.: Bridging patterns: An approach to bridge gaps between SE and HCI. In: Information and Software Technology, 48(2), 2006.

[7] Fowler, M.: Analysis Patterns: Reusable Object Models. Addison-Wesley, 1996.

[8] Hahsler, M.: Analyse Patterns im Softwareentwicklungsprozeß. PhD thesis at WU Wien, 2001.

[9] Müller, S.: Modellbasierte IT-Unterstützung von wissensintensiven Prozessen - Dargestellt am Beispiel medizinischer Forschungsprozesse. PhD thesis at the Universität Erlangen - Nürnberg, 2007.

[10] PLML, XML DTD available under http://www.hcipatterns.org/PLML+1.0.html

[11] Robertson, S.: Requirements trawling: techniques for discovering requirements. In: International Journal of Human-Computer Studies, Volume 55, Number 4, October 2001.

[12] Royce, W., W.: Managing the Development of Large Software Systems. In: Proceedings of the 9th international conference on Software Engineering. 1987.

[13] Scheer, A.-W.: CIOs entwickeln sich zu Chief Process Officers. In: CIO, 2007. http://www.cio.de/karriere/cios_im_portrait/810380/index4.html

[14] Shneiderman, B., Plaisant, C.: Designing the user interface. Addison Wesley, 2004.

[15] Schumacher, M., Fernandez, E., Hybertson, D., Buschmann, F., Sommerlad, P.: Security Patterns - Integrating Security and Systems Engineering. John Wiley & Sons, 2005.

[16] Siau, K., Ericksson, J., Lee, L.: Theoretical versus practical complexity: The case of UML. In: Journal of Database Management 16, 3, 2005.

[17] Völter, M., Stahl, T.: Model-Driven Software Development: Technology, Engineering, Management. Wiley, 2006.

[18] Zimmermann, B., Bergsträßer, S., Rensing, C., Steinmetz, R. A Requirements Analysis of Adaptations of Re-Usable (E-Learning) Content. In: In Proceedings of World Conference on Educational Multimedia, Hypermedia and Telecommunications 2006.

[19] Zimmermann, B., Rensing, C., Steinmetz, R.: Patterns for Tailoring E-Learning Materials to Make them Suited for Changed Requirement. Published in the Proceedings of VikingPLoP 2006.

[20] Zimmermann, B., Rensing, C., Steinmetz, R.: Patterns towards Making Web Material Accessible. Published in the Proceedings of EuroPLoP 2007.

14 Appendix

The Process Description Input Tool PIT supports adaptation process experts in creating process description of adaptation processes. It has been described in section 6. Figure 9 shows the main input form of PIT.

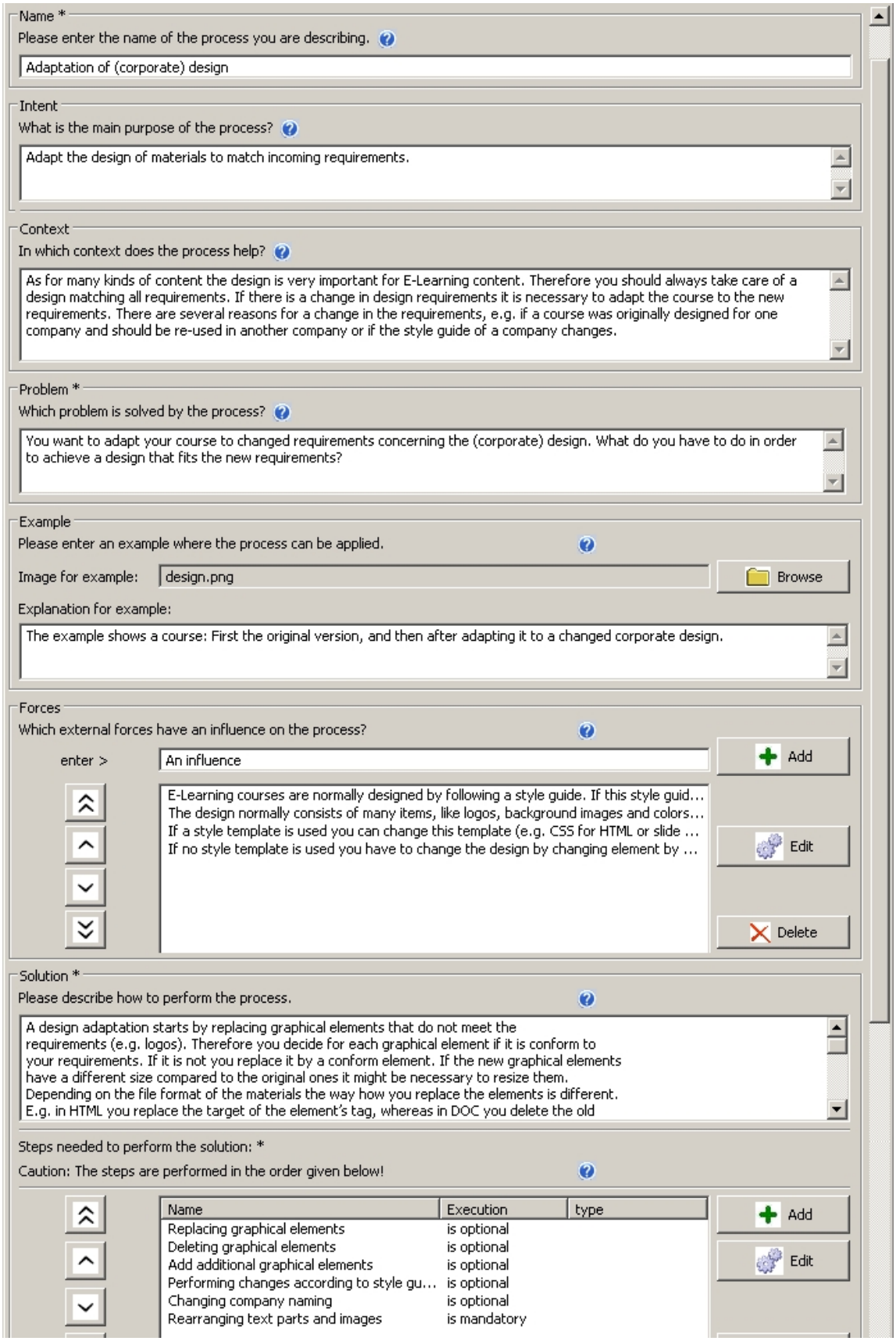


Figure 9: Part of PIT's input form.

PIT stores the process description given by the process experts in XML format (compare section 6). The following listing shows the DTD of the XML files containing the process information.

```
<!ELEMENT pattern (intent?, context?, problem, example_illustration?,
example_explanation?, forces, solution, process_steps, known_uses*,
consequences, related-patterns>
      <!ATTLIST pattern patternID ID #REQUIRED
                        confidence CDATA #IMPLIED
                        name CDATA #REQUIRED >
<!ELEMENT intent (#PCDATA)>
<!ELEMENT context (#PCDATA)>
<!ELEMENT problem (#PCDATA)>
<!ELEMENT example_illustration (#PCDATA)>
<!ELEMENT example_explanation (#PCDATA)>
<!ELEMENT forces (force*)>
<!ELEMENT force EMPTY>
      <!ATTLIST force name CDATA #REQUIRED>
<!ELEMENT solution (#PCDATA)>
<!ELEMENT Process_steps (Process_step+)>
<!ELEMENT Process_step EMPTY>
      <!ATTLIST Process_step
            name Name #REQUIRED
            mandatory (true | false) "true">
<!ELEMENT known_uses (#PCDATA)>
<!ELEMENT consequences
      (positive_consequence+, negative_consequence*)>
<!ELEMENT positive_consequence EMPTY>
      <!ATTLIST positive_consequence name CDATA #REQUIRED>
<!ELEMENT negative_consequence EMPTY>
      <!ATTLIST negative_consequence name CDATA #REQUIRED>
<!ELEMENT related_patterns (related_pattern*)>
<!ELEMENT Related_patterns (Related_pattern*)>
<!ELEMENT Related_pattern EMPTY>
      <!ATTLIST Related_pattern
            name CDATA #REQUIRED
            patternID ID #REQUIRED
            type CDATA #REQUIRED
      >
```

Listing 1: Pattern file DTD.

The information about the process flow is stored as a process graph. The following listing shows a part of such a process graph stored in XML (compare section 6.1). You can see a process identified via its ID. All process steps needed to perform the process are listed in the `requires` section of the process. For each process step it is noted if the step has to be performed (`mandatory="true"`) or not (`mandatory="false"`).

The last process step in the example is only performed if at least one of the steps before has been performed. Therefore all process steps, which can be performed before this step, are listed in the `precondition` section of the last process step. For the first process step you can see all atomic operations needed to perform this step. Again it is written down for each operation if the execution is mandatory. In addition you can see the three kinds of atomic operations (query, decision, and execution). The original file also contains a section defining the needed atomic operations. This section is not shown in the listing below.

```
<process id="process_pattern$56667" process-pattern="true">
  <requires fragmentRef="process_step$22357234" mandatory="false"/>
  <requires fragmentRef="process_step$25517184" mandatory="false"/>
  <requires fragmentRef="process_step$28757034" mandatory="false"/>
  <requires fragmentRef="process_step$36740146" mandatory="false"/>
  <requires fragmentRef="process_step$43532108" mandatory="false"/>
  <requires fragmentRef="process_step$50025522" mandatory="true">
    <precondition fragmentRef="process_step$22357234"/>
    <precondition fragmentRef="process_step$25517184"/>
    <precondition fragmentRef="process_step$28757034"/>
    <precondition fragmentRef="process_step$36740146"/>
    <precondition fragmentRef="process_step$43532108"/>
  </requires>
</process>
<process-step id="process_step$22357234">
  <requires functionRef="query$28693719" mandatory="true"/>
  <requires functionRef="decision$26294026" mandatory="true"/>
  <requires functionRef="decision$24376617" mandatory="false"/>
  <requires functionRef="query$23537464" mandatory="false"/>
  <requires functionRef="decision$32687500" mandatory="false"/>
  <requires functionRef="query$33442589" mandatory="false"/>
  <requires functionRef="execution$5472454" mandatory="false"/>
  <requires functionRef=" execution $51555041" mandatory="false"/>
</process-fragment>
```

Listing 2: Part of a process graph file.

Modeling Architectural Pattern Variants

Ahmad Waqas Kamal[1], Paris Avgeriou[1], and Uwe Zdun[2]

[1] Department of Mathematics and Computing Science,
University of Groningen, The Netherlands
a.w.kamal@rug.nl, paris@cs.rug.nl

[2] Distributed Systems Group,
Vienna University of Technology, Austria
zdun@infosys.tuwien.ac.at

Abstract Systematic modeling of architectural patterns is a challenging task mostly because of the inherent pattern variability and because pattern elements do not match the architectural abstractions of modeling languages. In this paper, we describe an approach for systematic modeling of architectural patterns using a set of architectural primitives and a vocabulary of pattern-specific architectural elements. These architectural primitives can be used as the basic building blocks for modeling a number of architectural patterns. We introduce profiles for the UML2 meta-model to express the architectural primitives. The use of the primitives along with the stereotyping scheme is capable of handling some of the challenges for the systematic modeling of architectural patterns, such as expressing pattern participants in software design.

keywords: Architectural Pattern, Architectural Primitive, Modeling, UML.

1 Motivation

Architectural patterns provide solutions to recurring problems at the architecture design level. These patterns not only document 'how' solution solves the problem at hand but also 'why' it is solved, i.e. the rationale behind this specific solution [14]. So far, a huge list of patterns has been documented in the literature [5,6]. These patterns have been successfully applied to design software in different domains and provide concrete guidelines for modeling the structural and behavioral aspects of software systems. Although at present, the practice of modeling architectural patterns is largely ad hoc and unsystematic, the topic of systematic pattern modeling is receiving increasing attention from researchers and practitioners [3].

Proceedings of the 13th European Conference on Pattern Languages of Programs (EuroPLoP 2008), edited by Till Schmmer and Allan Kelly, ISSN 1613-0073 (issn-1613-0073.html).

In spite of the benefits that patterns offer for solving recurring design problems and the ever-growing list of documented patterns, there is not yet a proven approach for the systematic modeling of architectural patterns and pattern variants in software design. Some architecture description languages (ADLs), such as UniCon [13], Aesop [8], ACME [9], and Wright [2] capture specific concepts for modeling patterns. However, none of the approaches presented so far, for modeling architectural patterns, can effectively express the semantics of architectural patterns [11]. This is because each pattern addresses a whole solution space comprised of different variants of the same pattern, which are difficult to express in a specific ADL. In contrast to ADLs, UML offers a generalized set of elements to describe software architecture but UMLs support for modeling patterns is weak because pattern elements do not match the architectural abstractions provided in UML. In summary, both ADLs and the UML provide only limited support for modeling patterns.

In our previous work [14], we identified a set of architectural primitives. These primitives offer reusable modeling abstractions that can be used to systematically model solutions that are repetitively found in different patterns. In this paper, we introduce a few more primitives and use all the primitives discovered during our current and previous work to devise an approach that is capable of systematically modeling architectural patterns in system design. The main contribution of this paper lies in modeling pattern variants using primitives, identifying pattern aspects that are difficult to express using primitives, and devising a generalized scheme that uses a vocabulary of pattern-specific components and connectors (e.g., pipes, filters) in conjunction with primitives for systematically modeling architectural patterns.

The remainder of this paper is structured as follows: In Section 2 we present our approach for representing patterns and primitives as modeling abstractions, exemplified using an extension of the UML. Section 3 briefly introduces the primitives discovered in our previous work while Section 4 gives detailed information of the new primitives documented in this paper. In section 5, we give an overview of the relationships between patterns and primitives. Section 6 describes the modeling of few selected pattern variants using primitives and a pattern-elements vocabulary. Section 7 compares related work and Section 8 discusses future work and concludes this study.

2 Extending UML to Represent Patterns and Primitives

UML is a widely known modeling language and is highly extensible [3]. There are two approaches for extending UML: extending the core UML metamodel or creating profiles which extend metaclasses. Our work focuses on the second approach where we create profiles specific to the individual architectural primitives. Although this work is exemplified using UML 2.0, the same approach can be used for other modeling languages as long as the selected modeling language supports an extension mechanism to handle the semantics of the primitives. The key idea is that a modeling language can be extended to facilitate semantics of the architectural primitives and that these primitives can then be used to model patterns.

We extend the UML metamodel for each discovered architectural primitive using UML profiles. That is, we define the primitive as extensions of existing metaclasses of the UML using stereotypes, tagged values, and constraints:

- *Stereotypes:* Stereotypes are one of the extension mechanisms to extend UML metaclasses. We use stereotypes to extend the properties of existing UML metaclasses. For instance, the Connector metaclass is extended to generate a variety of primitive-specific specialized connectors.
- *Constraints:* We use Object Constraint Language (OCL) [1] to place additional semantic restrictions on extended UML elements. For instance, constraints can be defined on associations between components, navigability, direction of communication, etc.

– *Tagged Values* allow one to associate tags to architectural elements. For example, tags can be defined to represent individual layers in a layered architecture using layer numbers.

We chose the UML profiles extension mechanism due to the following reasons:

– A large community of software architects understands UML as a software modeling language. This enables us to use the existing set of UML elements as the basis for extensions. Thus, the time needed to learn a new language and the risks of a novel approach are reduced.
– UML allows the creation of profiles without changing the semantics of the underlying elements of the UML metamodel. Profiles are good enough to serve for this purpose.
– A number of UML tools are available to design software architecture and support profiles out-of-the-box. In contrast, a metamodel extension would require an extension of the tools.

In the architectural primitives, presented in this paper, we mainly extend the following classes of the UML 2 metamodel to express the primitives:

– *Components* are associated with required and provided interfaces and may own ports. Components use connectors to connect with other components or with its internal ports.
– *Interfaces* provide contracts that classes (and components as their specialization) must comply with. We use the interface meta-class to support provided and required interfaces, where provided interfaces represent functions offered by a component and required interfaces represents functions expected by a component from its environment.
– *Ports* are the distinct points of interaction between the component that owns the ports and its environment. Ports specify the required and provided interfaces of the component that owns them.
– *Connectors* connect the required interfaces of one component to the provided interfaces of other matching components.

3 Architectural Primitives

This section provides an extension to our previous work [14] where we listed nine architectural primitives along with the mechanism to discover primitives in architectural patterns. We have used the same mechanism to discover new primitives in this paper. We first present five primitives discovered in the Component-Connector view that are repetitively found as abstractions in modeling variants of a number of patterns. Moreover some patterns documented in [7] are used as solution participants of other patterns, hence we consider their modeling solution as primitives and include them in our collection. Subsequently, in the next section, we extend the set of primitives with five new primitives.

Our original set of primitives was comprised of the following [14]:

– *Callback:* A component B invokes an operation on Component A, where Component B keeps a reference to component A in order to call back to component A later in time.
– *Indirection:* A component receiving invocations does not handle the invocations on its own, but instead redirects them to another target component.
– *Grouping:* Grouping represents a Whole-Part structure where one or more components work as a Whole while other components are its parts.
– *Layering:* Layering extends the Grouping primitive, and the participating components follow certain rules, such as the restriction not to bypass lower layer components.
– *Aggregation Cascade:* A composite component consists of a number of subparts, and there is the constraint that composite A can only aggregate components of type B, B only C, etc.

- *Composition Cascade:* A Composition Cascade extends Aggregation Cascade by the further constraint that a component can only be part of one composite at any time.
- *Shield:* Shield components protect other components from direct access by the external client. The protected components can only be accessed through Shield.
- *Typing:* Using associations, custom typing models are defined with the notion of super type connectors and type connectors.
- *Virtual Connector:* Virtual connectors reflect indirect communication links among components for which at least one additional path exists from the source to the target component.

4 Description and Modeling Solutions to Architectural Primitives in the Component-Connector View

In this section, we present five primitives that are repetitively found among a number of architectural patterns. For the first selected primitive, we briefly describe the primitive, discuss the issues of modeling the primitive in UML, present UML profile elements as a concrete modeling solution for expressing the primitive, and motivate known uses of the primitive in architectural patterns. For the sake of simplicity, the modeling issues and modeling solutions of remaining primitives are detailed in the Appendix.

4.1 Push-Pull

Context: Push, Pull, and Push-Pull structures are common abstractions in many software patterns. They occur when a target component receives a message on behalf of a source component (Push), or when a receiver receives information by generating a request (Pull). Both structures can also occur together at the same time (Push-Pull).

Modeling Issues: Semantics of push-pull structures are missing in UML diagrams. It is difficult to understand whether a certain operation is used to push data, pull data, or both. A major problem in modeling the patterns using Pushs or Pulls in UML is that although Push-Pull structures are often used to transmit data among components, it cannot be explicitly modeled in UML.

Modeling Solution: To properly capture the semantics of Push-Pull in UML, we propose a number of new stereotypes for dealing with the three cases Push, Pull, and Push-Pull. Figure 1 illustrates these stereotypes according to the UML 2.0 profile package, while Figures 2 and 3 depict the notation used for the stereotypes.

The Push-Pull primitive consists of the following stereotypes and constraints:

- *IPush*: A stereotype that extends the Interface metaclass and contains methods that Push data among components.
- *IPull*: A stereotype that extends the Interface metaclass and contains methods that Pull data among components.
- *PushPort*: A stereotype that extends the Port metaclass and is supported by IPush as provided interface and IPull as required interface. This can be formalized using two OCL constraints:

A Push port is typed by IPush as a provided interface

```
inv: self.basePort.provided->size() = 1
 and self.basePort.provided->forAll(
```

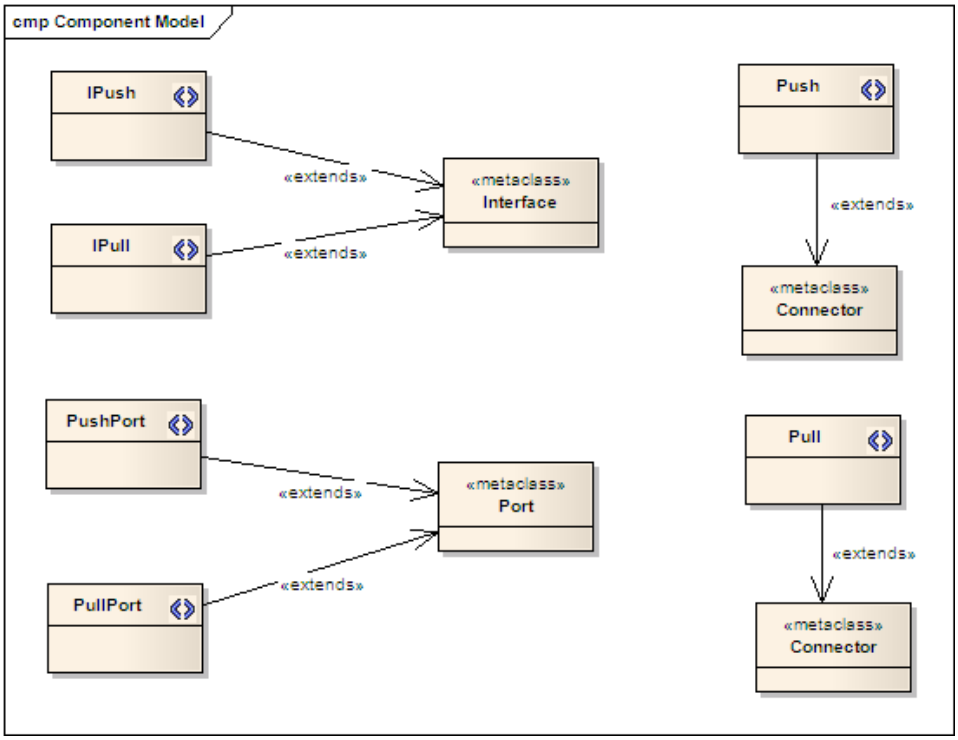


Figure 1. Stereotypes for modeling Push-Pull

```
    i:Core::Interface |
        IPush.baseInterface->exists (j | j=i))
```

A Push port is typed by IPull as a required interface

```
inv: self.basePort.required->size() = 1
 and self.basePort.required->forAll(
    i:Core::Interface |
        IPull.baseInterface->exists (j | j=i))
```

PullPort: A stereotype that extends the port metaclass and is supported by IPush as required interface and IPull as provided interface. This can be formalized using two OCL constraints for the Pull port:

A Pull port is typed by IPull as a provided interface

```
inv: self.basePort.provided->size() = 1
 and self.basePort.provided->forAll(
    i:Core::Interface |
        IPull.baseInterface->exists (j | j=i))
```

A Pull port is typed by IPush as a required interface

```
inv: self.basePort.required->size() = 1
 and self.basePort.required->forAll(
    i:Core::Interface |
        IPush.baseInterface->exists (j | j=i))
```

Push: A stereotype that extends the Connector metaclass and connects a PushPort with a matching PullPort of another component.

A Push connector has only two ends.

```
inv: self.baseConnector.end->size() = 2
```

A Push connector connects a PushPort of a component to a matching PullPort of another component. A PushPort matches a PullPort if the provided interface of the former matches the required interface of the later

```
inv: self.baseConnector.end->forAll(
e1,e2:Core::ConnectorEnd | e1 <> e2 implies (
    (e1.role->notEmpty() and
    e2.role->notEmpty()) and
    (if PushPort.basePort->exists(p |
    p.oclAsType(Core::ConnectableElement) =
    e1.role)
then
    (PullPort.basePort->exists(p |
    p.oclAsType(Core::ConnectableElement) =
    e2.role)
    and
    e1.role.oclAsType(Core::Port).required =
    e2.role.oclAsType(Core::Port).provided)
else
    PullPort.basePort->exists(p|
```

```
    p.oclAsType(Core::ConnectableElement) =
    e1.role)
endif)))
```

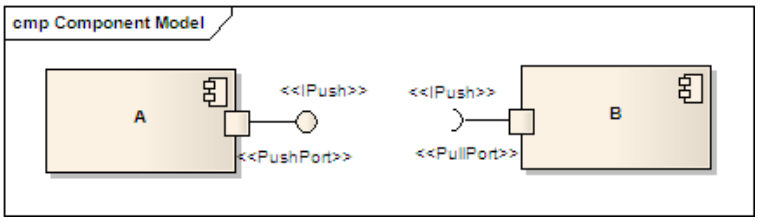


Figure 2. Ports and interfaces to model Push structure from B to A

Pull: A stereotype that extends the Connector metaclass and connects a PullPort with a matching PushPort of another component.

A Pull connector has only two ends.

```
inv: self.baseConnector.end->size() = 2
```

A Pull connector connects a PullPort of a component to a matching PushPort of another component. A PushPort matches a PullPort if the provided interface of the former matches the required interface of the later

```
inv: self.baseConnector.end->forAll(
e1,e2:Core::ConnectorEnd | e1 <> e2 implies (
(e1.role->notEmpty() and
e2.role->notEmpty() ) and
(if PushPort.basePort->exists(p |
p.oclAsType(Core::ConnectableElement) =
e1.role)
then
(PullPort.basePort->exists(p |
p.oclAsType(Core::ConnectableElement) =
e2.role)
and
e1.role.oclAsType(Core::Port).required =
e2.role.oclAsType(Core::Port).provided)
else
PullPort.basePort->exists(p|
p.oclAsType(Core::ConnectableElement) =
e1.role)
endif)))
```

Known uses in patterns:

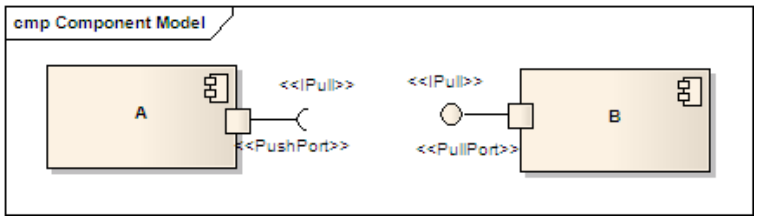


Figure 3. Ports and interfaces to model Pull structure from A to B

- In the Model-View-Controller [4] pattern, the model pushes data to the view, and the view can pull data from the model.
- In the Pipes and Filters [4] pattern, filters push data, which is transmitted by pipes to other filters. In addition, pipes can request data from source filters (Pull) to transmit it to the target filters.
- In the Publish-Subscribe [4] pattern, data is pushed from a framework to subscribers and subscribers can pull data from the framework.
- In the Client-Server [4] pattern, data is pushed from the server to the client, and the client can send a request to pull data from the server.

4.2 Virtual Callback

Context: Consider two components are connected via a callback mechanism. In many cases the callback between components does not exist directly, rather there exist mediator components between the source and the target components. Such information should be represented at the design level. For instance, in the MVC pattern, a model may call a view to update its data but this data may be rendered first by the mediator components before it is displayed on the GUI.

Modeling Issues: The virtual relationship is an important aspect to show collaborating elements. The standard UML supports connector or association links to model virtual relationships. However, such a relationship cannot be made explicit in standard UML as it may become difficult to determine which components have subscribed to other components to be called back virtually.

Modeling Solution: To capture the semantics of Virtual Callback properly in UML, we extend the Callback [14] primitive with constraints that a virtual callback can only be used between two components where there is a path of components and connectors that links A to B using following constraints:

To capture the semantics of callback primitive properly in UML, we use the following stereotypes: VirtualCallback, EventEnd, and CallbackEnd. The VirtualCallback extends the Connector metaclass while the EventEnd and CallbackEnd extend the ConnectorEnd metaclass where the EventOccurence takes place at the sender component (EventEnd) while the EventExecution takes place at the receiver end (CallbackEnd).

Known Uses in Patterns:

- In the MVC [4] pattern, the view and model components may communicate to each other virtually using callback operation.
- In the Observer [6] pattern, the subjects may observe the target objects virtually.
- In the Publish-Subscribe [4] pattern, the publishers may callback subscribers virtually.

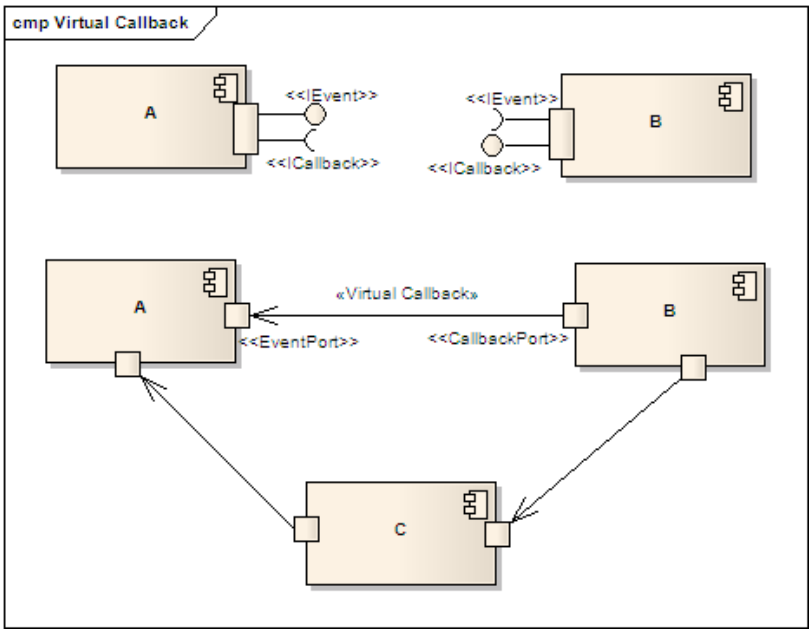


Figure 4. The notation of the stereotypes in Virtual Callback Modeling

4.3 Delegation Adaptor

Context: This primitive converts the provided interface of a component into the interface the clients expect. The Delegation Adaptor primitive is a close match to the Object Adaptor [**?**] pattern.

Modeling Issues: Adaptors shield the underlying system implementation from its surroundings. However, adaptors can not be explicitly modeled using the architectural abstractions present in UML as their task is more focused on conversion rather than computation.

Modeling Solution: To capture the semantics of Adaptor properly in UML, we propose the following new stereotypes: AdaptorPort extends the Port metaclass and is typed by the IAdaptor as provided interface and IAdaptee as required interface. Both the IAdaptor and IAdaptee stereotypes extend the Interface metaclass.

Known Uses in Patterns:

- In the Layers [4] pattern, the adaptor supports the separation of explicit interface of a layer from its implementation.
- In the Broker [4] pattern, the adaptor translates the messages coming from remote services to the underlying system.
- In the Microkernel [4] pattern, the adaptor is used to map communication between external and internal servers.
- In the Proxy [6] pattern, the adaptor is used to separate the interface from the implementation.

4.4 Passive Element

Context: Consider an element is invoked by other elements to perform certain operations. Passive elements do not call operations of other elements.

Modeling Issues: UML components do not structurally differentiate between active and passive elements. Such a differentiation is important to understand clearly the responsibility of individual elements in the design.

Modeling Solution: To capture the semantics of Passive Element properly in UML, we use the following new stereotypes: PElement extends the Component metaclass and attaches the PassivePort. The IPassive stereotype extends the Interface metaclass and types the PassivePort, which extends the Passive metaclass.

Known Uses in Patterns:

- In the Pipes and Filters [4] pattern, the passive filter cannot pull or push data to its neighboring filters.
- In the MVC [4] pattern, the passive view only receives or displays data to the user and does not invoke any operation on a model or controller elements.
- In the Client-Server [4] pattern, the passive server does not invoke any operation on client-side and responds only to the client requests.

4.5 Interceder

Context: Sometimes certain objects in a set of objects cooperate with several other objects. Allowing direct link between such objects can overly complicate the communication and result in strong coupling between objects [6]. To solve this problem, Interceder components are used.

Modeling Issues: Interceder components are typically involved in decoupling components and store the collective behavior of interacting components. The structural representation of mediator components in UML diagrams is hard to understand.

Modeling Solution: To capture the semantics of Interceder primitive properly in UML, we propose following new stereotypes: Incdr, IncdrPort, and IFIncdr. Incdr extends the Component metaclass and attaches IncdrPort. IncdrPort extends the Port metaclass and is typed by the provided interface IFIncdr.

Known Uses in Patterns:

- In the PAC [4] pattern, a controller is used to intercede communication between agents in the PAC hierarchy.
- In the Microkernel [4] pattern, an interceder component receives requests from external server and dispatches these requests to one or more internal servers.
- In the Reflection [4] pattern, the meta level components intercede communication by providing interfaces to facilitate modification in underlying components.

5 The Pattern-Primitive Relationship

Architectural patterns and architectural primitives are complementary concepts. Modeling patterns in a system design is applying one of the alternate solutions to solve specific problems at hand [4] where as primitives serve as the building blocks for expressing architectural patterns. In this context, patterns offer general solutions while primitives offer relatively more specific solutions. Similar to the selection of architectural patterns among complementary patterns, primitives might also need to be selected among complementary primitives, e.g., based on the system requirements you might choose either Shield or Indirection. Such a decision to select the appropriate primitive involves the context in which the pattern is applied, and the specific solution variant addressed by the pattern. Moreover, certain primitives can be used in combination with

Patterns \ Primitives	Callback	Indirection	Grouping	Layering	Aggregation Cascade	Composition Cascade	Shield	Typing	Virtual Connector	Push	Pull	Virtual Callback	Adaptor	Passive Element	Control	Interceder
Active Repository [5]	X										X	X				
Broker [4]		X	X				X			X						
Cascade [6]					X	X										
Client Server [4]	X		X	X						X	X	X	X			
Component and Wrapper [5]		X									X					
Composite [6]					X	X										
Facade [6]		X	X				X									
Event [6]	X															
Explicit Invocation [6]	X															
Indirection Layers [5]			X	X	X			X		X	X					
Interceptor [6]	X															
Interpreter [6]		X									X					
Knowledge Level [5]								X								
Layered System [5]		X			X					X	X		X			
Layers [4]			X	X			X			X	X				X	
Message Redirector [5]		X					X									
Microkernel [4]		X										X				
MVC [4]	X									X	X	X			X	
Observer [6]	X										X					
Object System Layer [5]					X		X	X								
Organization Hierarchy [5]					X	X										
PAC [4]	X														X	X
Pipes and Filters [4]										X	X			X		
Proxy [4]		X					X									
Publish Subscribe [4]	X									X	X	X	X			
Reactor [6]	X															
Reflection [4]				X								X				
Remote Proxy [5]									X							
Type Object [5]								X								
Virtual Machine [5]		X	X				X									
Visitor [6]	X															
Wrapper Facade [5]		X									X					

Table 1. Patterns to Primitives mapping (X: found in documented pattern)

other primitives. For example, the Callback and Push-Pull primitives can work in conjunction to serve a common purpose.

Table 1 provides a patterns-to-primitives mapping, which is based on the primitives discovered so far in our work. The detailed discussion about the discovery of each primitive in the related patterns is already documented in the Known Uses in Patterns subsections of our current and previous work (see Section 3 and [14] for details). The intention is to use the pool of all available primitives to model several architectural patterns. However, the mapping from patterns to primitives is not one-to-one: rather different variants of the patterns can be modeled using a different combination of primitives. Thus, the decision to apply a specific primitive for modeling patterns lies with the architect who selects primitive(s) that best meet the needs to model the selected pattern(s).
The issues addressed above directly deal with the traditional challenge of modeling pattern variability. The solution variants entailed by a pattern can be applied in infinite different ways and so is the selection of primitives for modeling pattern variants. More important is that whichever pattern variant is applied in system design, it should address the solution clearly with structural and semantic presence. Using our primitives allows an architect to apply a near infinite solution variants with certain level of reusability. Such a reusability support also depends on the context in which the pattern is applied as in some cases extra constraints or missing pattern semantics may be required.

5.1 Expressing Missing Pattern Semantics in UML

An important aspect of modeling architectural patterns is the explicit demonstration of patterns in system design and support for automated model validation. Such a representation helps in better understanding of the system by allowing the user to visualize and validate the patterns. The primitives described above capture recurring building blocks found in different patterns. However, it may be the case that certain pattern aspects of a specific solution variant may not be fully expressed by the existing set of primitives. Therefore, for expressing missing pattern semantics that are not covered by the primitives, we provide support to the user with a vocabulary of design elements that can be used alongside with the primitives to fully express pattern semantics such as pipes, filters, client, server etc. For this purpose, we define few stereotype in UML with known semantics of the selected architectural patterns. For instance, a component can be stereotyped as filter and a connector can be stereotyped as pipe. The stereotyping scheme presented here is further complimented by using these stereotypes for modeling the example patterns in the next section.

The use of pattern-specific design elements for expressing pattern variants has a number of significant benefits. First, it offers reusability support for expressing patterns in system design. The well-known properties entailed by documented pattern variants can be reapplied in system design as a solution to new problems. Second, this makes it easier for a stakeholder to understand design of the system. For example, the use of design vocabulary to express pipes and filters in system design makes an architecture more explicit to understand and the way different architectural elements fit in the structure. Third, it offers a good support for automated model validation by ensuring that selected patterns are correctly applied in a system design. All of these three benefits compliment our use of primitives for modeling patterns. The intention is that though primitives offer good reusability and model validation support, as advocated in our current and previous work [14], the stereotyping scheme presented in this section makes the story complete for the systematic modeling of architectural patterns and pattern variants.

6 Modeling Architectural Patterns Using Primitives

In this section, we use the primitives and stereotyping scheme described in the previous sections to model specific pattern variants. The patterns modeled in this section are specialization to the patterns documented in POSA [4] and hence are called pattern variants. We do not claim to cover all the variability aspects of the selected patterns. However, an effort to describe selected pattern variants using primitives provides a solid base for modeling unknown pattern variants as well. To serve this purpose, we have selected three traditional architectural patterns namely the Layers, Pipes and Filters, and Model-View-Controller (MVC). We use the following guidelines to model each selected pattern variant:

- A brief description of selected pattern variants
- Mapping selected pattern variants to the list of available primitives
- Highlight the issues in modeling pattern variants using primitives
- Use stereotyping scheme to capture the missing pattern semantics.

6.1 Pipes and Filters

The Pipes and Filters pattern consists of a chain of data processing filters, which are connected through pipes. The filters pass the data output to the adjacent filters through pipes. The elements in the Pipes and Filters pattern can vary in the functions they perform e.g. pipes with data buffering support, feedback loops, forks, active and passive filters etc. The primitives discovered so far address many such variations for systematically modeling Pipes and Filters pattern. However, certain aspects of the Pipes and Filters pattern may not be fully expressed by the primitives e.g. feedback loops, forks, etc. The requirements we consider in this section for modeling the specific Pipes and Filters pattern variant are: a) filters can push or pull data from the adjacent filters; b) filters can behave as active or passive elements; and c) feedback loop.

At first, we map the selected Pipes and Filters pattern variant to the list of available primitives. We select the Push, Pull, and Passive Element primitives from the existing pool of primitives. The rationale behind the selection of these primitives is as follows:

- The Push and Pull primitives are used to express the pipes that transmit streams of data between filters.
- The filters that are not involved in invoking any operations on their surrounding elements are expressed using the Passive Element primitive.

Missing Semantics: As described in section four, the challenge to model missing pattern semantics is solved by stereotyping UML elements. In the current example, the selected primitives are sufficient to express the Push, Pull, and Passive Elements in the Pipes and Filters pattern. However, we identify that the feedback loop cannot be fully expressed using the existing set of primitives. The existing primitives can express that the data is pushed or pulled between the filters but this does not express the presence of a feedback structure. Similarly, the semantics of the Pipe and Filter elements are not applied using the existing set of primitives.

Additional Stereotypes: We apply the Feedback stereotype on the Push primitive to capture the structural presence of feedback loop in the Pipes and Filters pattern. Such a structure represents that the data is pushed from one filter to another filter using the feedback loop. The original Push primitive, as described in section four, extends the UML metaclasses of connector, interface, and port. While the feedback stereotype further specializes the Push primitive by labeling

it as Feedback. The introduction of feedback stereotype does not introduce new constraints nor affects the underlying semantics of the Push primitive. Figure 5 shows the stereotypes used for expressing Pipes and Filters pattern.

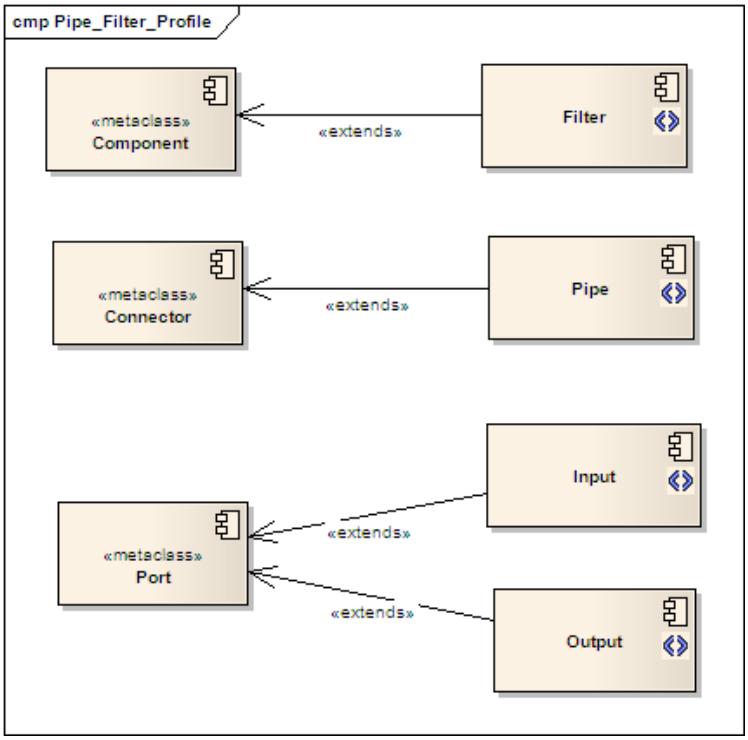


Figure 5. UML stereotypes for expressing Pipes and Filters pattern participants

Feedback: A stereotype that is applied to the Push primitive for expressing the Feedback structure in the Pipes and Filters pattern variant. Feedback stereotype extends the Connector metaclass of UML.

The second stereotype named Filter that we use from the existing vocabulary of design elements is defined as follows:

Filter: A stereotype that extends the Component metaclass of UML and attaches input and output ports.

A Filter component is formalized using the following OCL constraints:

An Input port is typed by Iinput as a provided interface

```
inv: self.basePort.provided->size() = 1
 and self.basePort.provided->forAll(
    i:Core::Interface |
       Iinput.baseInterface->exists(j | j = i))
```

An Output port is typed by Ioutput as a required interface

```
inv: self.basePort.required->size() = 1
    and self.basePort.provided->forAll(
        i:Core::Interface |
            Ioutput.baseInterface->exists(j | j = i))
```

The third stereotype that we use from the existing vocabulary of design elements is Pipe that is defined as follows:
Pipe: A stereotype that extends the Connector metaclass of UML and connects the output port of one component to the input port of another component.
A Pipe is formalized using following OCL constraints:

```
inv: self.baseConnector.end->size() = 2
```

As shown in figure 6, the first filter in the chain works as a passive filter and does not invoke any operations on its surrounding filters. While the second filter is an active filter, which pulls data from the passive filter and after processing pushes this data to the next filter in the chain. The third filter in the chain sends data back to the passive filter for further processing, and sends the final processed data to the sink.

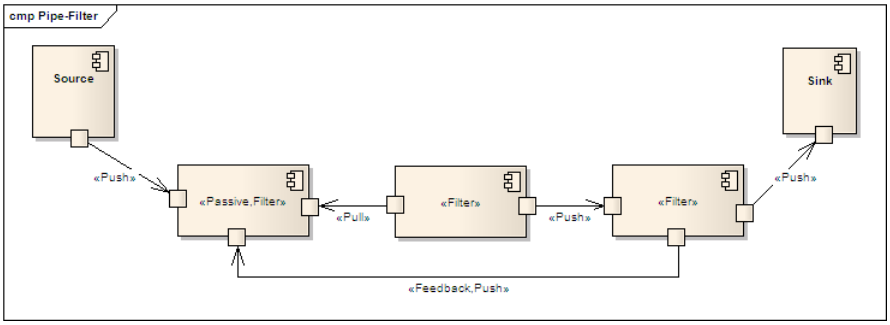


Figure 6. Modeling Pipes and Filters Pattern Variant Using Primitives

6.2 Model-View-Controller

The structure of the MVC pattern consists of three components namely the Model, View, and Controller. The Model provides functional core of an application and notifies views about the data change. Views retrieve information from the Model and display it to the user. Controllers translate events into requests to perform operations on View and Model elements. Usually a change propagation mechanism is used to ensure the consistency between the three components of the MVC pattern [4].

As a first step, we map the MVC pattern to the list of available primitives as shown in the table in section four. We select the Callback, Passive Element and Control primitives for modeling the MVC pattern. The rationale behind the selection of these primitives is as follows:

- The View subscribes to the model to be called back when some data change occurs and works as passive object by not invoking any operation on the Model.

Missing Semantics: However, not every aspect of the MVC pattern can be modeled using the existing set of primitives. For instance, the Model, View, and Controller components are not mapped to any primitives discovered so far. Keeping in view the general nature of these components, there is a need to provide reusability support by including these three pattern elements in the existing vocabulary of design elements.

Additional Stereotypes: As described above, despite the reusability support offered by the selected primitives, the MVC pattern semantics are not structurally distinguishable. We use the following three stereotypes from the existing set of design elements:

Model: A stereotype that extends the Component metaclass of UML and attaches ports for interaction with the Controller and View components.

Controller: A stereotype that extends the Component metaclass of UML and attaches ports for interaction with the Model and View components.

View: A stereotype that extends the Component metaclass of UML and attaches ports for interaction with the Model and Controller components.

As shown in Figure 7, the Controller receives input and translates it into requests to the associated model using the Control primitive. While, the Model calls back View when a specific data change occurs.

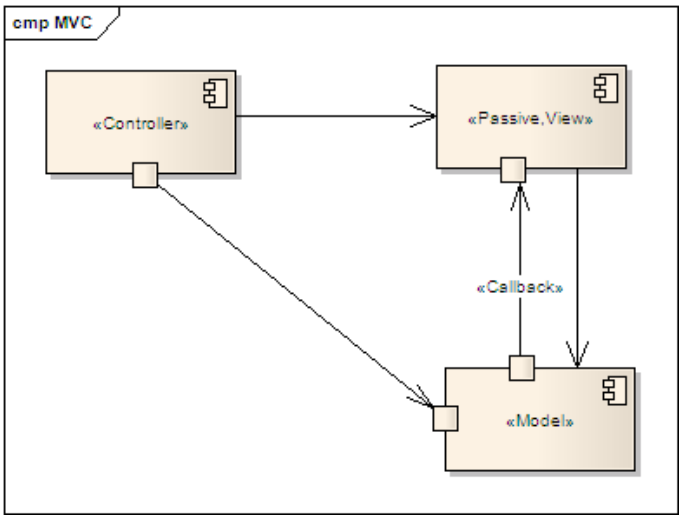


Figure 7. Modeling MVC Pattern Using Primitives

6.3 Layers

The Layers pattern groups elements at a certain level of abstraction where lower layers provide services to the adjacent upper layer. Such a structure is used to reduce dependencies between objects in different Layers. As a first step, we map the Layers pattern to the list of available primitives and select the Layering primitive. The rationale behind the selection of this primitive is as follows:

- Components are members of specific layers where each lower layer provides services to the adjacent upper layer
- A component can only be a member of one layer

Missing Semantics: In the Layers pattern, the high-level functions implementation relies on the lower level ones. Such system requires horizontal partitioning where each partition carries operations at a certain level of abstraction. As each layer in the Layers pattern is a virtual entity so it cannot exist without the presence of at least one component. Moreover, the upper layers cannot bypass the layers for using services in the bottom layers i.e. in Figure 8, the group members from layer3 can call components in layer2, but not into layer1.

Additional Stereotypes: Almost all structural characteristics of the Layers pattern are modeled using the Layering primitive as shown in Figure 8. Using the layering primitive, the constraints assure that within an individual layer all component work at the same level of abstraction and no component belongs to more than one layer at any time. Moreover, no additional stereotyping of UML elements is required to model this specific variant of the layers pattern.

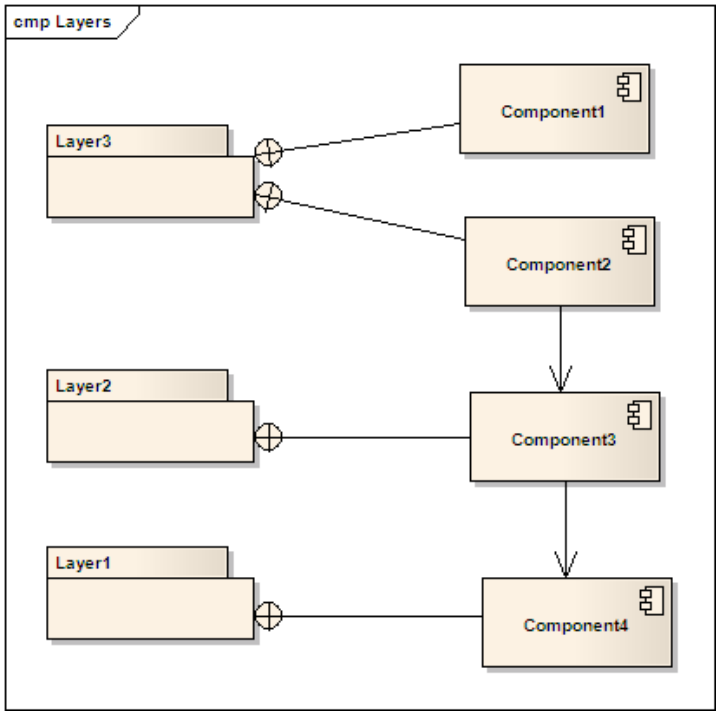


Figure 8. Modeling Layers Pattern Using Primitives

7 Related Work

The approach described in this paper is based on our previous work [14] where we present an initial set of primitives for modeling architectural patterns. However, the idea to use primitives

for software design is not novel and has been applied in different software engineering disciplines [12]. The novelty of our work lies in the use of primitives for systematically modeling architectural patterns, which has not be addressed before.

Using different approaches, a few other researchers have been working actively on the systematic modeling of architectural patterns [8]. Garlan et al. [8] proposes an object model for representing architectural designs. The authors characterize architectural patterns as specialization of the object models. However, each such specialization is built as an independent environment, where each specialization is developed from scratch using basic architectural elements. Our approach significantly differs in a way that our focus is on reusing primitives and pattern elements and only where required we extend the primitives and pattern elements to capture the missing pattern semantics.

Simon et al. [10] extends the UML metamodel by creating pattern-specific profiles. The work by Simon et al. maps the MidArch ADL to the UML metamodel for describing patterns in software design. However, this approach does not address the issue of modeling a variety of patterns documented in the literature rather manual work is required to create profiles for each newly discovered pattern. Our approach distinctively differs from this work as we focus on describing a generalized list of patterns using the primitives.

Mehta et al. [12] propose eight forms and nine functions as basic building blocks to compose pattern elements. Their approach focuses on a small set of primitives for composing elements of architectural styles. Our approach is different in the sense that we offer a more specialized set of primitives that are captured at a rather detail level of abstraction. Moreover, we use vocabulary of pattern elements in parallel to architectural primitives to capture the missing semantics of architectural patterns.

8 Conclusion

Using architectural primitives and pattern-specific design elements vocabulary in combination offers a systematic way to model patterns in system design. We have extended the existing pool of primitives with the discovery of five more primitives. With the help of few examples, we show an approach for modeling architectural pattern variants using primitives. The scheme to use stereotyping in conjunction with primitives offers: a) reusability support by providing vocabulary of design elements that entail the properties of known pattern participants; b) automated model validation support by ensuring that the patterns are correctly modeled using primitives; and c) explicit representation of architectural patterns in system design.

To express the discovered primitives and design elements vocabulary, we have used UML2.0 for creating pattern-specific profiles. As compared with the earlier versions, UML2.0 has come up with many improvements for expressing architectural elements. However, we still find UML as a weak option in modeling many aspects of architectural patterns e.g. weak connector support. As a solution to this problem, the extension mechanisms of the UML offers an effective way for describing new properties of modeling elements. Moreover, the application of the profiles to the primitives allows us to maintain the integrity of the UML metamodel. By defining primitive-specific profiles, we privilege a user to apply selective profiles in the model.

As future work, we would like to advance in the automation of our approach by developing a tool, which supports modeling pattern variability, documentation, analyzing the quality attributes, source code generation, etc. We believe that we can discover more primitives in different architectural views in near future, which will provide a better re-usability support to architects for systematically expressing architectural patterns.

References

1. Object constraint language specification. *OMG Standard*, 1.1.
2. R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, Volume 6, No. 3(ACM Transactions on Software Engineering and Methodology):213–249, 1997.
3. M. Bjorkander and C. Kobryn. Architecting systems with uml 2.0. *IEEE Softw.*, 20(4):57–61, 2003.
4. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. Wiley & Sons, 1996.
5. D. C. S. Frank Buschmann, Kevlin Henney. *Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing*. Wiley Series in Software Design Patterns, 2007.
6. D. C. S. Frank Buschmann, Kevlin Henney. *Pattern-Oriented Software Architecture: On Patterns and Pattern Languages*. Wiley Series in Software Design Patterns, 2007.
7. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
8. D. Garlan, R. Allen, and J. Ockerbloom. Exploiting style in architectural design environments. *SIGSOFT Softw. Eng. Notes*, 19(5):175–188, 1994.
9. D. Garlan, R. Monroe, and D. Wile. Acme: an architecture description interchange language. In *CASCON '97: Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research*, page 7. IBM Press, 1997.
10. S. Giesecke, F. Marwede, M. Rohr, and W. Hasselbring. A style-based architecture modelling approach for uml 2 component diagrams. In *Proceedings of the 11th IASTED International Conference Software Engineering and Applications (SEA'2007)*, pages 530–538. ACTA Press, Nov. 2007.
11. A. W. Kamal and P. Avgeriou. An evaluation of adls on modeling patterns for software architecture design. In *4th International Workshop on Rapid Integration of Software Engineering Techniques*, 26 November 2007.
12. N. R. Mehta and N. Medvidovic. Composing architectural styles from architectural primitives. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 347–350, New York, NY, USA, 2003. ACM.
13. M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Trans. Softw. Eng.*, 21(4):314–335, 1995.
14. U. Zdun and P. Avgeriou. Modeling architectural patterns using architectural primitives. *Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 133–146, 2005.

9 Appendix

For the architectural primitives presented in this work, following we provide the OCL constraints used to express the semantics of architectural primitives precisely in a system design.

9.1 Virtual Callback

We use the following OCL constraints to define the semantics of callback primitive:

```
inv: self.baseConnector.end.role.oclAsType(
 Core::Property).class->forAll(
```

```
    c1,c2:Core::Component | c1 <> c2 implies
        c1.oclAsType(Core::Component).connects(c2))
```

9.2 Delegation Adaptor

To capture the semantics of Adaptor properly in UML, we use the following OCL constraints:

AdaptorPort is typed by IAdaptor as a provided interface

```
inv: self.basePort.provided->size() = 1
 and self.basePort.provided->forAll(
    i:Core::Interface |
        IAdaptor.baseInterface->exists (j | j=i))
```

AdaptorPort is typed by IAdaptee as a required interface

```
inv: self.basePort.required->size() = 1
 and self.basePort.required->forAll(
    i:Core::Interface |
        IAdaptee.baseInterface->exists (j | j=i))
```

AdapteePort is typed by IAdaptee as a provided interface

```
inv: self.basePort.provided->size() = 1
 and self.basePort.provided->forAll(
    i:Core::Interface |
        IAdaptee.baseInterface->exists (j | j=i))
```

AdapteePort is typed by IAdaptor as a required interface

```
inv: self.basePort.required->size() = 1
 and self.basePort.required->forAll(
    i:Core::Interface |
        IAdaptor.baseInterface->exists (j | j=i))
```

Adaptor component attaches the AdaptorPort.

```
inv: self.baseComponent.ownedPort.name = 'AdaptorPort'
```

9.3 Passive Element

To capture the semantics of Passive Element properly in UML, we use the following OCL constraints:

PassivePort provides the IPassive interface

```
inv: self.basePort.provided->size() = 1
 and self.basePort.provided->forAll(
    i:Core::Interface |
        IPush.baseInterface->exists (j | j=i))
```

PElement attaches the PassivePort

inv: self.baseComponent.ownedPort.name = 'PassivePort'

9.4 Interceder

To capture the semantics of Interceder primitive properly in UML, we use the following OCL code:

A IncdrPort is typed by IRIncdr as a provided interface

```
inv: self.basePort.provided->size() = 1
 and self.basePort.provided->forAll(
    i:Core::Interface |
       IRIncdr.baseInterface->exists (j | j=i))
```

A IncdrPort is typed by IFIncdr as a provided interface

```
inv: self.basePort.provided->size() = 1
 and self.basePort.provided->forAll(
    i:Core::Interface |
       IFIncdr.baseInterface->exists (j | j=i))
```

An Interceder component owns IncdrPort

```
inv: self.baseComponent.ownedPort.name = 'IncdrPort'
```

Junkies Like Us

How the Social Web Influences Our Understanding Of Privacy

Andreas Rüping
Sodenkamp 21 A, 22337 Hamburg, Germany
andreas.rueping@rueping.info
www.rueping.info

Introduction

With Web 2.0 becoming increasingly popular, people tend to make more and more personal information available on the Internet: in social networks, blogs, chats and wikis. The younger generation especially seems to enjoy social networks and gives information about their lifestyle away freely. Have we, or will we, become Web junkies who deliberately put more or less their entire lives online?

Looking at the material that people put online, it's easy to be torn between two positions:

- On the one hand, there's the concept of online communities, the idea of a democratic Web, the perspective of a more open society, and the fun that comes from actively participating in today's most popular medium.
- On the other hand, there's the unpleasant prospect of endless personalised advertising, the possible danger that virtually anyone can track you down and collect arbitrary information about you, and the possibility that all our concerns for privacy might vanish one day.

So is the Social Web a good thing or a bad?

This focus group set out to discuss this question. We first analysed the benefits and risks involved in the Social Web and then moved on to explore possible strategies and personal practices for handling the challenges. The following photograph shows the session output that was produced.

Proceedings of the 13th European Conference on Pattern Languages of Programs (EuroPLoP 2008), edited by Till Schümmer and Allan Kelly, ISSN 1613-0073 <issn-1613-0073.html>.

Evaluation

Benefits

There was consensus among the focus group participants that there is value in the Social Web and that it makes options available that weren't available before. The following table summarises the benefits of the Social Web that were identified during the session.

Area	*Benefit*
Access to information and services	• gain access to information (event announcements, etc.) through web sites, blogs, etc. • gain access to services (shopping, banking, printing services)
Publishing general information	• share / publish opinions and recommendations (for books, music, movies, restaurants) • share / publish photos • tag information with keywords • rate published material
Publishing personal information	• build an information repository (bookmarks, etc.) • share / publish personal status / location • share / publish personal photos

Community building	• stay in touch • build a community of practice • build a niche community
Culture	• satisfy your curiosity • build collective intelligence • add diversity to the web • improve free speech in totalitarian countries / repressive societies

Borderline Properties

In addition, the focus groups participants identified a few characteristics of the Social Web that they were reluctant to classify as benefits, although these characteristics aren't necessarily negative either. They were tentatively named 'borderline properties'. The following table summarises these 'borderline properties' — things that can be regarded as positive or negative, depending on perspective.

Area	*Benefit*
Information transparency	• employer checking out a job candidate's personal data • candidate putting material online to improve job chances • easier investigation into crime, etc.

Risks

Just as there was no doubt about the Social Web offering benefits, there was no doubt about the existence of risks either. Participants' opinions varied regarding how severe the specific risks are, but it was clear from the discussion that awareness of these risks is a precondition for using the Social Web safely and successfully. The following table summarises the risks that were identified.

Area	*Risk*
Information overload	• a flood of useless information (requires effective filters) • relative importance of search engines and information portals ("if you can't find it, it doesn't exist")
Lack of reliability	• amateurisation (everybody acts as a journalist) • subjective information is confused with facts • intentional / unintentional misinformation

Lack of authority	• unclear copyrights • plagiarism • derivative work (users who copy from different sources and publish under their own name)
Lack of awareness	• information is less protected than users think • illusion of anonymity
Data misuse	• user monitoring (without the user knowing) • cross-linking (information about a user being collected from different sources) • personal data (etc. addresses, profile, personal preferences) being sold to third parties (etc. for personalised advertising)
User misbehaviour	• open doors for slander without legal redress • vandalism / personal threats (especially when anonymous)
Time	• published material is volatile • no information revocation (once published, information cannot be deleted)
Culture	• communication stress (pressure to be always online) • exhibitionism • growing disregard of privacy (pressure to put private information online)

Strategies and Practices

The second half of the focus group was devoted to a discussion of possible strategies for handling the challenges imposed on us by the Social Web. The discussion was somewhat controversial, but anyway we were able to come up with a list of strategies, or personal practices, that were widely regarded as useful.

Some of the strategies may seem slightly contradictory at first (especially the first two in the following table), but in fact they aren't. They aim to resolve the tension built up by the Social Web, its benefits and risks, and their combination should constitute a sensible approach to using the Social Web.

The following table lists these strategies formulated as prototypical patterns.

In a way, others expect you to put information online and if you don't, then that will give a poor impression of you, therefore: Tell the world what you want the world to know. Put material online if you're sure it represents you well.
Once you've published something, everyone can (in principle) read it and you can't delete it either, therefore: Apply selective foresight. Publish material only if it's ok with you if the world finds out about it.
Sharing (with a community) and publishing (for everyone to see) may not be the same thing, but information can leak, therefore: Share information only if it's generally ok should the information get published.
Web 2.0 is full of unreliable sources, therefore: Question your sources. Watch out for opinions dressed as facts. Look up multiple references.
Some parts of Web 2.0 are more reliable than others, some are perhaps dangerous. Smaller (more specific) communities are often more trustworthy, therefore: Build a mental map of the Web and identify areas of trust. Find out about the people behind the scenery (owners of a social site etc.).
Some sites respect your privacy more than others. Some are quite ok with respect to privacy, while others have made selling your personal data a part of their business model, therefore: Favour sites with an opt-in policy (where by default, personal information may not be passed to third parties) over sites with an opt-out policy (where users must actively deselect the dissemination of their data).
More and more stuff is put online and it's easy to lose track, therefore: Reflect regularly about your sharing habits. Google yourself regularly.

Conclusion

Web 2.0 is a pretty cool place. All focus groups participants said they used it to some extent, although there are clear limits to what we would put online and to what online platforms we'd use for sharing information with others.

But it's not just us that matters. Almost all participants agreed that, if they were 15 years younger, they'd probably use the Social Web more. Perhaps much more. It's safe to assume that there are (younger) people out there who use the Social Web quite extensively, largely oblivious to the existing risks. We therefore felt it was important to name these risks and to think of ways to avoid them.

The idea is not to avoid the Social Web. The idea should to embrace it, and at the same time to be aware of its technological background, its dangers and its cultural implications. Hopefully the strategies developed in this focus group can contribute to this goal.

Acknowledgements

Thanks to all participants for their contributions and for turning this focus group into three hours of fruitful discussion. Special thanks to Ademar Aguiar for taking the photograph and for providing the reference to [Doctorow 2007].

Suggested Reading

[Anderson 2006]

Chris Anderson. *The Long Tail - Why The Future of Business Is Selling Less Of More*. Hyperion, 2006.

A book that explains how the Internet has helped niche markets to come to fruition. Not primarily about the Social Web, but significant anyway.

[Berners-Lee 2008]

Tim Berners-Lee. *Questions & Answers*. BBC News, 2008-Mar-17.

An interview with the founder of the Web who argues against net tracking.

[Doctorow 2007]

Cory Doctorow. *Little Brother*. "www.craphound.com/littlebrother/", 2007.

A novel about a teenage hacker's quest against omnipresent surveillance.

[Hodgkinson 2008]

Tom Hodgkinson. *With Friends Like These...* The Guardian, 2008-Jan-18.

A newspaper artcile that details reservations about a popular Social Web site.

[Shirky 2008]

Clay Shirky. *Here Comes Everybody - The Power of Organizing Without Organizations*. Penguin, 2008.

A book about the Social Web and its implications on our society. An in-depth analysis of what is described as one the most revolutionary developments of our time.