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.