“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.

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.

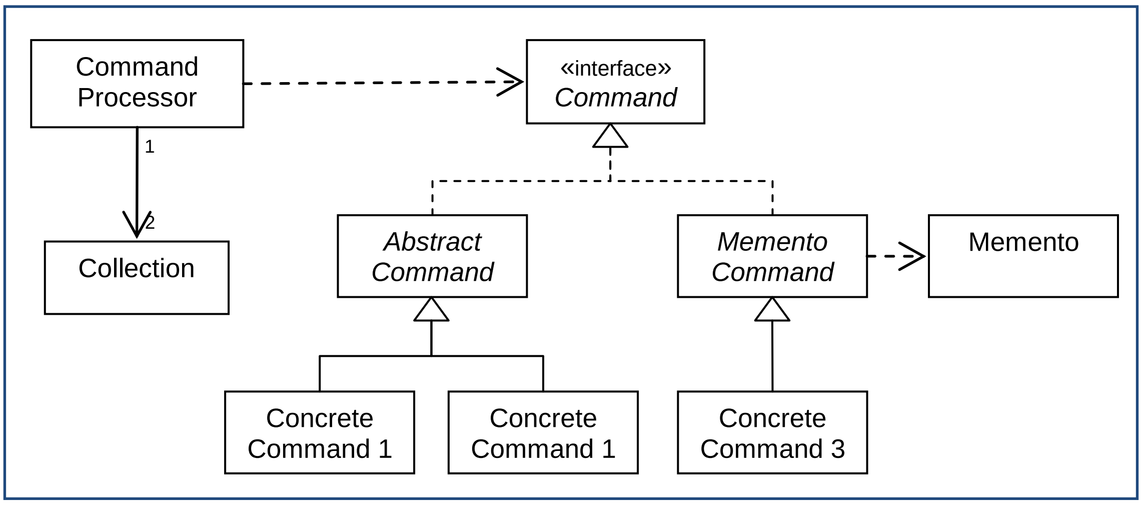
After introducing the undo mechanism, you recognise that there is also a need for a redo facility, to allow previously undone COMMAND objects to be re-executed. You need to determine how the COMMAND PROCESSOR can best accommodate both undo history and redo futures for COMMAND objects.

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.

View (1/18)
Your order will arrive at 15:35. If this is not convenient please ring before 15:15
Options
OK
Reply

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.

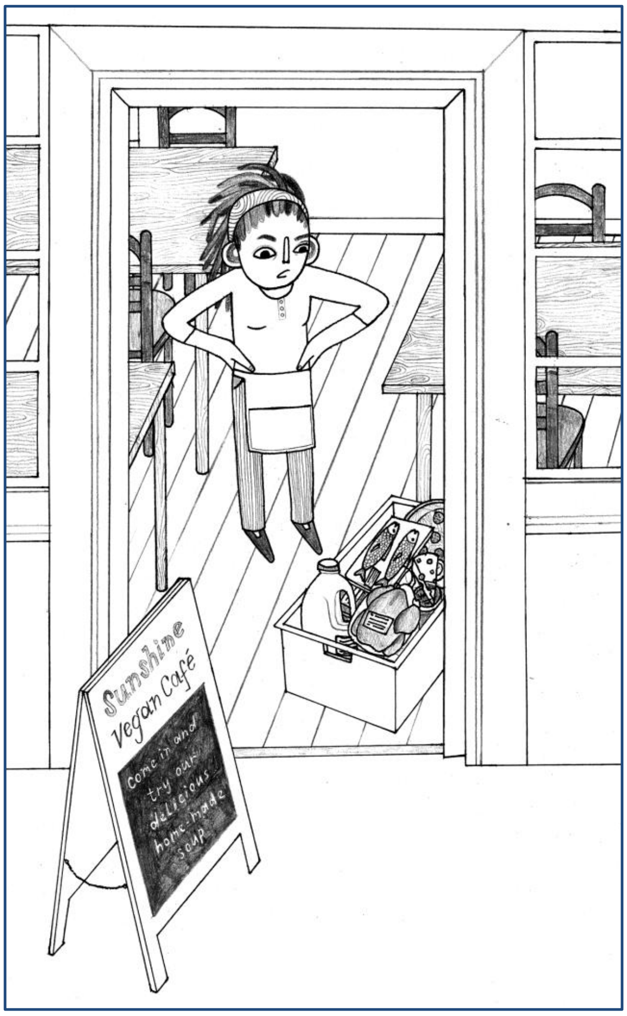
Turn to 5.

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 C*OMMAND*
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.

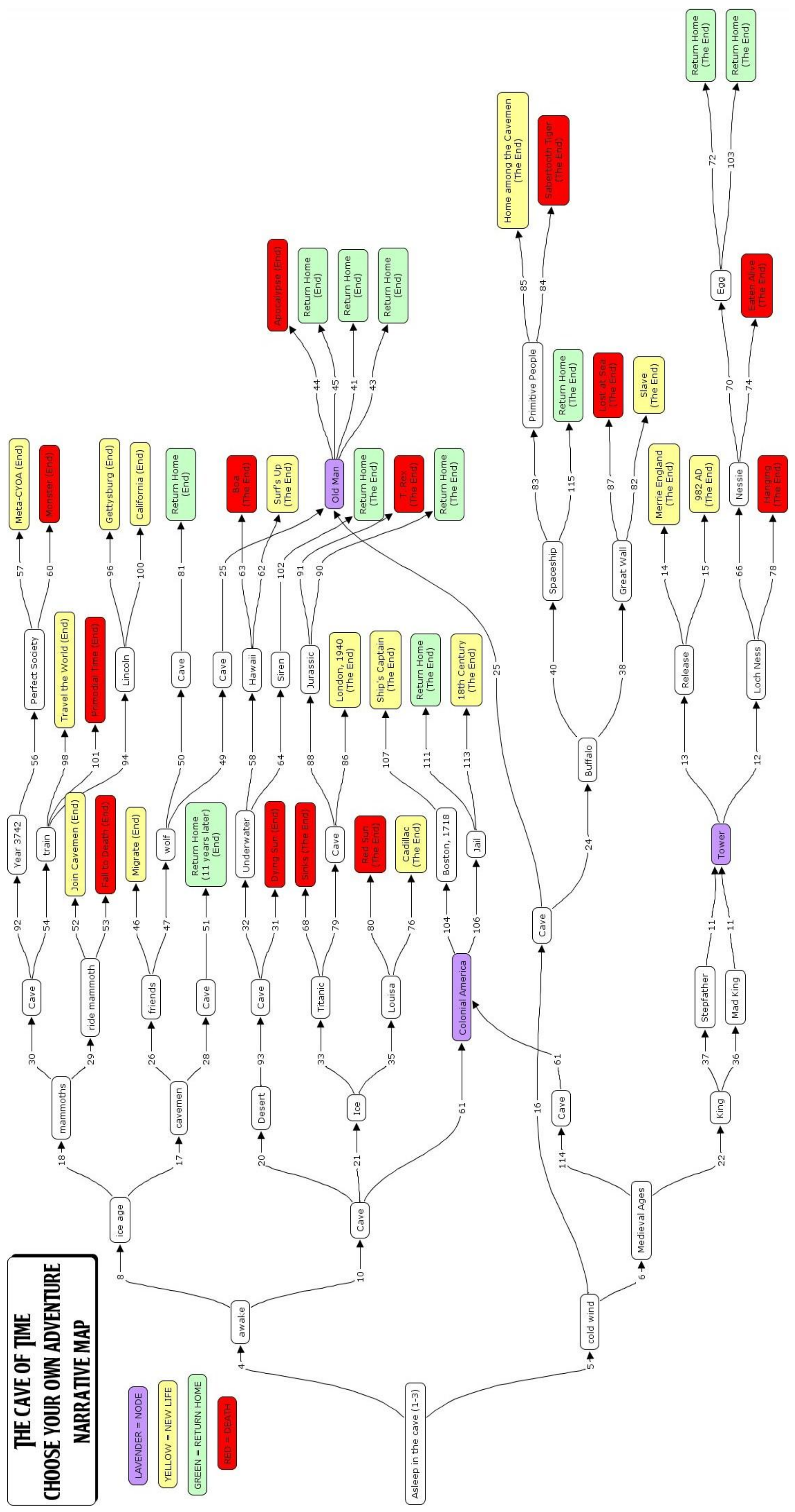


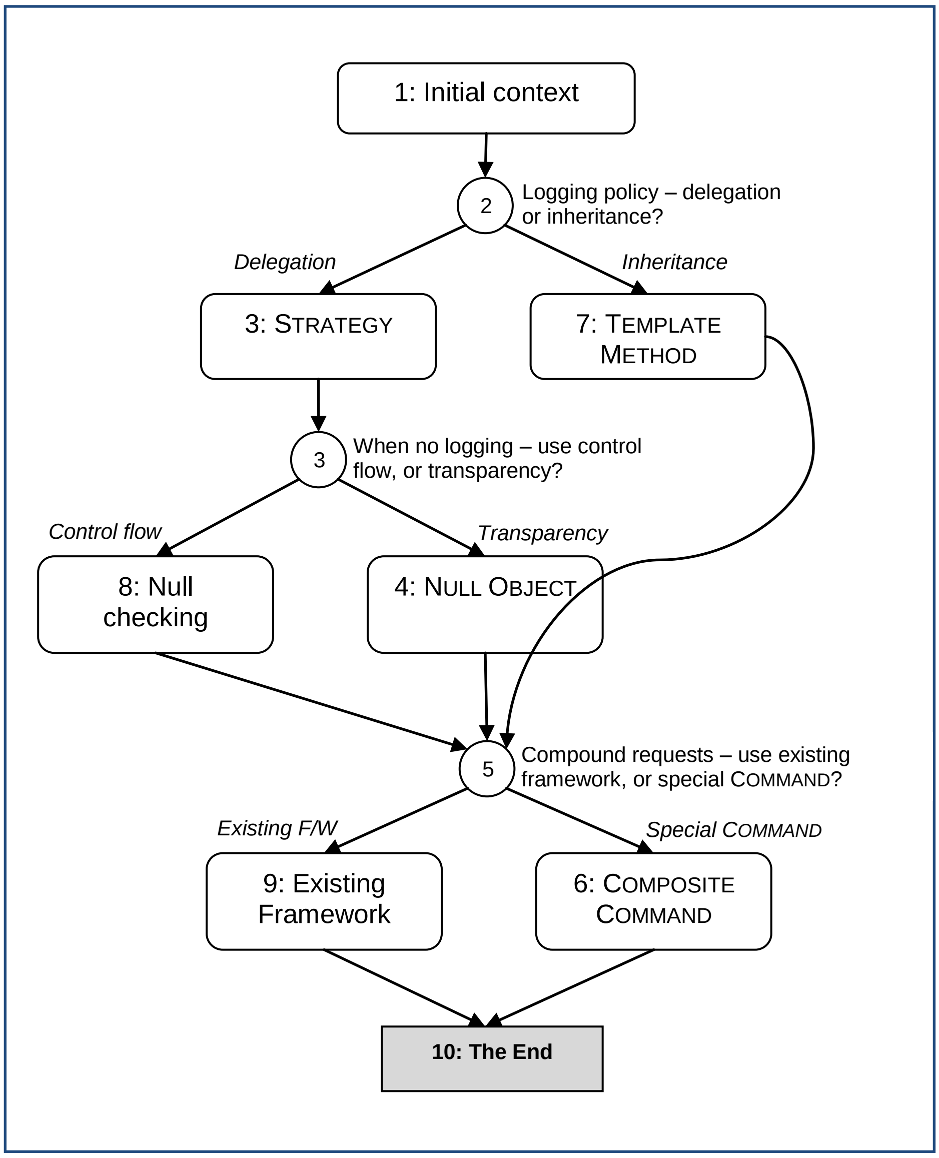
Figure 1- Map courtesy of Assistant Prof. Mark Sample from George Mason University, built using CMap [CMap].

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.

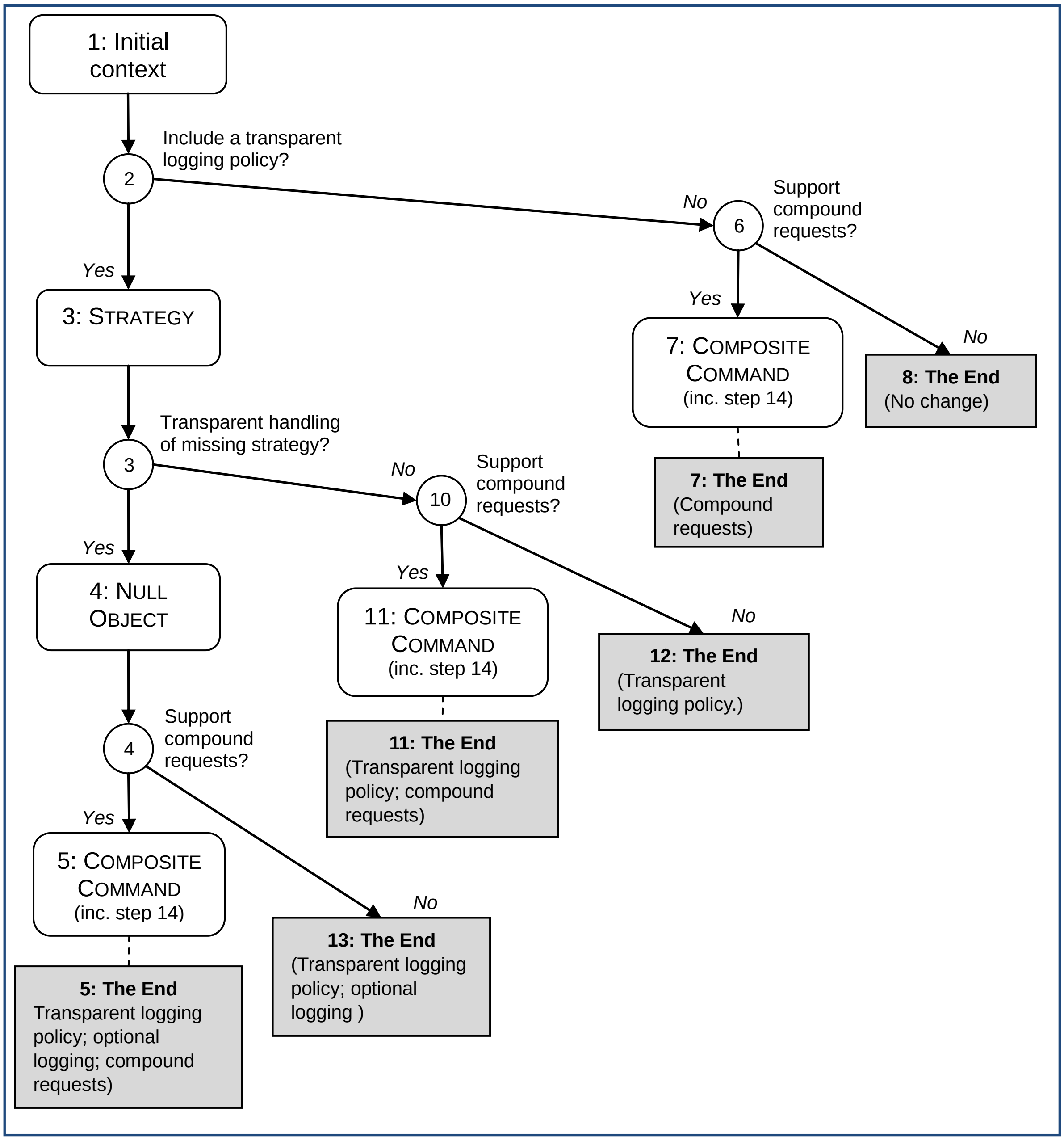
In each diagram, circles represent decisions points and italicised text shows possible choices. Rounded boxes represent the resulting development activities and decision consequences, and numbers denote discrete steps in the interactive story. Where numbered steps include both development activities 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



1: Initial context
2
Include a transparent logging policy?
Yes
No
3: STRATEGY
6
Support compound requests?
Yes
No
7: COMPOSITE COMMAND (inc. step 14)
7: The End (Compound requests)
8: The End (No change)
3
Transparent handling of missing strategy?
Yes
No
4: NULL OBJECT
10
Support compound requests?
Yes
No
11: COMPOSITE COMMAND (inc. step 14)
11: The End (Transparent logging policy; compound requests)
12: The End (Transparent logging policy.)
4
Support compound requests?
Yes
No
5: COMPOSITE COMMAND (inc. step 14)
5: The End Transparent logging policy; optional logging; compound requests)
13: The End (Transparent logging policy; optional logging)

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