"Choose Your Own Architecture" - Interactive Pattern Storytelling

James Siddle[1] and Maisie Platts[2]

[1] Independent, jim@jamessiddle.net,
WWW home page: http://www.jamessiddle.net
[2] Independent, maisie_platts@yahoo.co.uk
WWW home page: http://maisieplatts.com/

Abstract. The concept of Interactive Pattern Stories is introduced as a way to support software design education. An example interactive pattern story is presented, along with benefits, liabilities, and applicability of the approach. Key benefits include enabling readers to explore different choices to design problems and to experience positive and negative consequences of design choices, and the engaging game-like format. The key liability is the complexity of the writing task. The main application area is to education and learning.

1 Introduction

> *"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)?"* - Step 1 of "The Warlock of Firetop Mountain" [1]

This paper proposes the concept of *interactive pattern stories*, as a way of supporting the exploration of pattern-based designs in an engaging, educational, and fun way. The *"Choose Your Own Adventure"* [2] style of book is proposed as a suitable basis for introducing interactivity into *pattern stories*.

An example interactive pattern story, presented below, is used to show the benefits of this medium to software design education. The interactive story, based on a previously published story, is interactive around design alternatives, illustrates the consequences of different design choices, and allows exploration pattern-based designs.

The rest of this paper is structured as follows. The target audience is introduced, followed by an introduction to pattern and interactive fiction concepts. The origin and structure of the interactive story is then described along with the key benefits offered to software design education. Reader guidance provides essential information to interactive story readers, then the interactive story itself

appears. Next, an analysis of story features, benefits, liabilities, and applicability of the approach is presented. The paper closes with an overview of related and further work, and conclusions.

1.1 Target Audience

Computer science students, software developers, and software architects will gain insight into the application of patterns, choices available during software design, and will learn several desirable and undesirable design choices related to 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 benefit from learning an interactive approach to describing the design and development of software through patterns.

2 Concepts

2.1 Patterns, Pattern Stories, Pattern Languages

A *pattern* [3] is a solution to a problem that occurs in a particular context, captured in an easy to understand format. A *pattern story* [4] describes the application of one or more patterns. Pattern stories can be derived from *pattern languages* [3], 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* [4] 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 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.

Note however that patterns and 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, leaving 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 his or her knowledge of the specific problem being faced to fill in the gaps in any particular pattern.

2.2 "Choose Your Own Adventure" and Interactive Fiction

"Choose Your Own Adventure" [2] 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 many endings, some good, others bad. For an in-depth examination of interactive fiction see "Twisty Little Passages: An Approach to Interactive Fiction" by Nick Montfort [5].

3 Interactive Pattern Stories

3.1 Origin of the Story

This paper presents an interactive story based heavily on a "request handling" pattern story published in *"Pattern-Oriented Software Architecture, Volume 5: On Patterns and Pattern Languages"* [3] [6]. 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 to solve the problems. This pattern story was originally derived from the pattern language published in [7].

Rather than write a completely new interactive story from scratch, the request handling story was transformed into the interactive version that appears below. The text was reworded into second-person (a defining characteristic of interactive fiction stories), and decisions and associated consequences were introduced into the tail end of the story.

3.2 Story Structure

The story is structured around decisions that the reader makes, relating to the Strategy, Template Method, Null Object, and Composite Command patterns.

The narrative, design choices, and consequences in different story paths are derived from variations to the request handling story suggested by the story authors (see [8]), as well as pattern descriptions and connections found in the pattern language in [7].

Additionally, the story is based around a fixed set of functional and quality requirements, which the reader is expected to fulfil. In the story, the functional requirements - a system's capabilities, services, and behaviour [9] - must always be fulfilled. The quality requirements - the qualities of the system being developed that are influenced by design decisions taken [9] - vary according to the reader's choices.

3.3 Key Benefits of the Approach

The benefits of the approach to readers are:

[3] Frank Buschmann, Kevlin Henney, Douglas C. Schmidt. Copyright 2007, John Wiley & Sons Limited. Reproduced with permission.

- The ability to explore different solutions to design problems.
- They can experience both positive and negative consequences of design choices.
- They will be engaged by the game-like format.

These benefits are explored in section 6.3 below.

4 Guidance for the Interactive Story Reader

4.1 Requirements

The interactive story is based around two functional requirements, and three quality requirements. The functional requirements will always be fulfilled, whilst the quality requirements may or may not be fulfilled based on your actions. The text of the story describes the consequences of your design decisions in relation to the quality requirements.

You may wish to refer back to this point when you are presented with choices in the story to refresh your memory.

Functional Requirements

- **Requirement 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.
- **Requirement F2:** The ability to create compound requests, to support composition of commands that have been written to be processed by the framework.

Quality Requirements

- **Requirement Q1:** Developers and users of the framework should find it easy to work with (*understandability*).
- **Requirement Q2:** It should be easy to perform routine maintenance of framework and framework-using code, such as fault correction or performance improvement (*maintainability*).
- **Requirement Q3:** It should also be easy to take advantage of new software or hardware technologies that may become available in the future (*evolvability*).

Note that functional requirement references are denoted below with an *F*, quality requirements with a *Q*.

4.2 How to Read the Story

Start reading at step 1, which provides the context for the story[4].

Make sure you are familiar with the requirements presented above, then simply follow the decision instructions as they appear. A route map for the story can be found in the appendices, along with thumbnails for each pattern used.

A few other things to bear in mind are:

- The decisions presented are intentionally short on information to keep each story succinct, and to promote exploration of the design options. Under ideal circumstances, design decisions would be based on an assessment of all relevant information, 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 - most software projects employ some form of source control, allowing earlier versions of source code to be reverted to. Please take this as an implicit option that simplifies the presentation of available options.
- Similarly, the choices presented do not represent the entire set of decisions available, rather a subset chosen to enable exploration of software design in the particular context. In reality, software professionals are always free to make whatever choice they wish. More experienced or advanced practitioners may find the choice constraints limiting.

[4] 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 story.

5 The Interactive Story

Step 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.

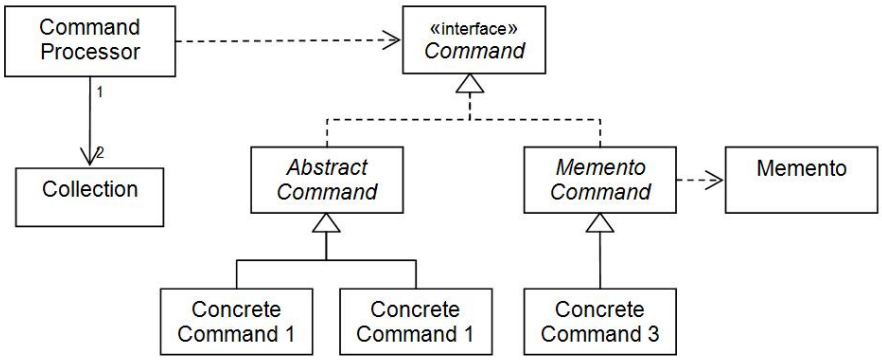

Fig. 1. UML diagram of the software described so far

The UML diagram in figure 1 shows the software decribed so far.

Now continue at step 2...

Step 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.

Step 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.

Step 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.

Step 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.

Step 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 [10] implementations can be complex and non-obvious.

Turn to 10.

Step 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.

Step 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.

See figure 2 for a real world example of the consequences of your decision.

Fig. 2. Step 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.

Turn to 5.

Step 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.

Step 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

6 Analysis

Below, the features of the interactive story are discussed, and different paths through the story are compared. The benefits and liabilities of the approach are then examined, along with related and future work.

6.1 Interactive Story Features

Alternative Decision Points At step 2 of the story the reader is presented with design alternatives that allow the choice of differing but equally desirable solutions to the problem - one choice leads to TEMPLATE METHOD, ther other to STRATEGY, both reasonable solutions given the context.

Optimal versus Sub-optimal Decision Points The story also allows the reader to explore the negative consequences that may be encountered if the desirable solution for the context (i.e. the 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.

Simple Decision Descriptions The decision descriptions are intentionally brief, omitting many important details. The intention of this approach is to encourage the reader to explore all possible paths; decision texts provide just enough information to make a choice, but not enough that the 'right' choice is immediately obvious. Similarly, patterns are described through the story, but it is left to the reader to learn more through the thumbnails in the appendix, and associated references.

Joining Branches The interactive story branches, but rejoins 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 is unaffected by differences introduced by 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. Note, however, that it may not always be possible to reconcile story branches.

Story Ending Step 10 concludes the story by summarising the functional requirements the reader fulfilled, and by prompting the reader to assess their design. The ending is intentionally vague and unrelated to the design choices taken; this is 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. An alternative would be to present different endings depending on the reader's choices (see Further Work, below).

Illustrations The story also includes an illustration associated with a particular story step. This acts to tie the reader's decisions to real world consequences, illustrating possible consequences of the reader's choices, and engaging the reader.

6.2 Comparison of Alternative Paths

To understand the interactive nature of the story, consider the following paths:

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 is that 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.

This highlights the purpose of the story - to encourage the reader to learn about design by through exploration.

6.3 Benefits and Liabilities

Benefits As mentioned above, benefits of the approach are that readers: have ability to explore different solutions to design problems; can experience both positive and negative consequences of design choices; will be engaged through the game-like format.

The decision making mechanism allows readers to explore various pattern-based designs possible for a particular set of requirements, and to experience negative consequences of sub-optimal choices. Going down the 'wrong' path gives the reader an understanding of negative consequences, but with no risk. Cheating is to be encouraged - after going down the wrong path, the reader can backtrack and change their mind, exposing them to the positive consequences of other choices. Subsequent readings of significantly different routes, such as those relating 'horror story' designs, may give the reader further insight.

Simple decision descriptions that omit information supporting reader choices, such as which choice is optimal, benefit the reader by encouraging the exploration of both optimal and sub-optimal paths. This encourages the reader to think 'outside the box', widening their exposure to all possible design consequences. However this approach may not always be appropriate because some readers may feel the game is 'rigged'. Similarly the omission of complete pattern descriptions may be confusing to some readers. A solution to both these problems is to ensure the reader is well-equipped with references to supporting material before they begin, along with guidance on how to approach making decisions and how to absorb the story.

Where interactive story paths are derived from a pattern language, the reader will gain an understanding of the overall context, problems and solutions, and pattern relationships in the language.

The format is engaging because reader decisions affect the outcome. The story takes on a game-like element where the set of outcomes is constrained by the reader's choices, providing an engaging, fun experience.

Further, interactive stories in the "Choose Your Own Adventure" format are written in a second person, genderless way. This avoids the dry, uninteresting tone of 'third person passive' writing. The authors of *"Pattern-Oriented Software Architecture: Volume 5"* [11] 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 much more engaging. Illustrations that show real world consequences of the reader's actions (as found in most children's interactive fiction) further engage the reader.

Liabilities The main liability of the approach is the complexity of the writing task. Even writing the simple story above was non-trivial, requiring many different possibilities to be considered and accounted for. Interactive pattern stories are also difficult to modify after creation.

The complexity of the writing task suggests the approach is better suited to academic and educational fields than industrial projects, though starting with the pattern story from [11] and the pattern language in [7] simplified the writing process considerably. Tooling may also increase the feasibility of the approach, for example the Storyspace [5] or iWriter [6] tools may simplify story development.

The decisions in the example story also lack any choices around whether to fulfil requirements or not. Educational use of interactive pattern stories may require both design alternative and requirement fulfilment choices.

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.

Finally, an interactive pattern story could be applied in a prescriptive way to limit design options, for example to force designers to always use STRATEGY to support transparent logging policies. This is likely to be unwelcome and would be considered a 'strait-jacket', unnecessarily restricting design choices.

6.4 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.

Different audiences may 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, different aspects of software design and development may be illuminated.

The addition of explicit back-tracking options may be useful for certain audiences, for example where the writer wishes to ensure the reader will reach a certain story path (optimal or otherwise). The addition of code or UML 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 linear narrative stories, which may be desirable for some readers. Interactive stories written with tooling support would be suitable candidates for generating such linear stories, as long as tooling supported such functionality.

[5] Storyspace website: `http://www.eastgate.com/Storyspace.html`
[6] iWriter, by talkingpanda software: `http://talkingpanda.com/iwriter/`

It may also be possible to employ the approach for 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 industrial application or technical documentation because of the effort involved in creating and updating the stories. Again, tool support may make such applications feasible.

6.5 Related Work

Patterns, pattern languages, pattern sequences, and pattern stories can all be used for software design education, and are related to interactive pattern stories as follows.

Patterns provide examples of good solutions to design problems, study of which provides readers with an understanding of principles behind the good solution, and exemplary solutions which can serve as the basis of future designs [10]. Patterns provide the problems and good solutions for 'optimal' interactive story steps, suggest less optimal story steps that may occur if a pattern is not applied, and serve as the underlying descriptions of good design choices.

Pattern languages connect individual patterns together to form a broader guidance framework targetting a particular problem domain. The study of pattern languages provides an understanding of problems that occur in that domain, the patterns to apply to solve those problems, and how patterns are related [11]. Pattern languages can provide the overarching context for interactive stories, suggest potential story directions following story steps that describe applying a pattern, and provide underyling pattern descriptions targetting a particular domain.

Pattern sequences describe particular paths through a pattern language; a sequence describes a combination of patterns, solving related problems in a particular domain, which are known to have successfully created a good design for that domain [4]. Pattern sequences provide complete, optimal, story paths for an interactive pattern story, which must be filled in with details and augmented with alternatives to create a full interactive story.

Finally pattern stories provide concrete examples of one or more patterns in action [4]. A pattern story may be derived from a pattern sequence, or may simply tell the story of several patterns that were applied together to solve related design problems. The study of pattern stories allows readers to understand the concrete application of one or more patterns in the real world, filling in the gaps in the more abstract patterns, languages, and sequences. Pattern stories may used as the basis of interactive pattern stories, as is the case for the interactive story above.

The interactive pattern story concept builds on the pattern concept, may be applied in support of pattern languages and sequences, and is closely related

to the pattern story concept. The key differences between pattern stories and their interactive form are: reader choices; the potential inclusion of sub-optimal design descriptions; and the engaging, educational format provided by interactive fiction.

6.6 Further Work

The biggest challenge to the successful application of interactive pattern stories is in authoring. The example presented above is simple enough to illustrate the benefits of interactive pattern stories; but stories developed for real-world use are likely to be more complex, and this carries a risk for writers in being overwhelmed by complexity. In particular the introduction of multiple endings is known to be problematic, requiring overlapping narrative in different story steps, and occasionally artificial design choices to provide complete coverage of all options. Further work is required to find a simple, accessible story format and structure that avoids accidentally complexity, allows exploration of design choices, and keeps the writer focussed on the story rather than the mechanisms of its telling.

A second area of further work is to test interactive pattern stories in the real world. While an informal workshop at ACCU 2009[7] supports the benefits outlined here, a more formal study is needed. The creation and application of one or more interactive pattern stories with a group of student volunteers is one option.

Other areas of exploration are the application of tool based approaches to both story writing and telling, and the use of other media such as board games or card based games (e.g. STRATEGY trumps TEMPLATE METHOD[8]).

6.7 Conclusions

This paper proposed the introduction of interactivity into pattern stories to engage readers and support the exploration of pattern-based designs and pattern languages for educational purposes. The *"Choose Your Own Adventure"* game-book format was proposed as a suitable basis for introducing interactivity.

An example interactive story was used to show the benefits of the proposed medium to software design education. In the story, the reader was able to explore design alternatives in solving a fixed set of functional requirements, where consequences were described in relation to several quality requirements.

The benefits of the approach are that readers can explore different solutions to design problems, that readers can experience both positive and negative consequences of design choices, and that the reader is engaged by the game-like format. The liabilities are the complexity of the writing task, the possibility

[7] Workshop on "Exploring Design Space with interactive pattern stories", James Siddle and Kevlin Henney, ACCU 2009.

[8] Top Trumps official website: `http://www.toptrumps.com/`

of pattern misapplication, and the fact that prescriptive stories may be unwelcome. The approach was considered to be applicable primarily in educational environments.

Further work is needed to find a simple, accessible story format that shields the writer from complexity but provides readers with the ability to explore design choices and consequences.

6.8 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 of Thames Hudson for providing excellent feedback and guidance on the layout of the paper. Thanks to Oxford University's Kellogg college for providing funding for me to attend EuroPLoP 2008. Finally thank you to the reviewers and editors of Transactions on Pattern Languages of Programs for supporting the development of the final version of this paper.

7 Appendices

7.1 Appendix - Pattern Thumbnails

For the purposes of this paper the patterns used in the interactive story are paraphrased below with references to the suitable pattern descriptions:

Command [10] 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 [7] To enable component reuse, whilst avoiding unnecessary coupling to component internals, separate the declared interface of a component from its implementation.

Command Processor [7] 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 [7] 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 [10] 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 [10] 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 [10] 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 [12] 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 [11] 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.

7.2 Appendix - Story Map

Figure 3 provides an overview of the decisions that you can make and the different routes through the interactive story found in this paper.

Circles represent decisions points and italicised text shows possible choices; rounded boxes represent the resulting development activities and decision consequences; numbers denote discrete steps in the interactive story. Where the numbered steps describe both development activities and choices, the numbers are repeated in the diagram. The grey box represents text that summarises the story at the end.

References

1. Jackson, S., Livingstone, I.: The Warlock of Firetop Mountain. 25th anniversary edn. Wizard Books (August 2007)
2. Packard, E.: Choose Your Own Adventure 1: The Cave of Time. Bantam Books (1979)
3. Alexander, C., Ishikawa, S., M. Silverstein, e.a.: A Pattern Language. Oxford University Press (1997)

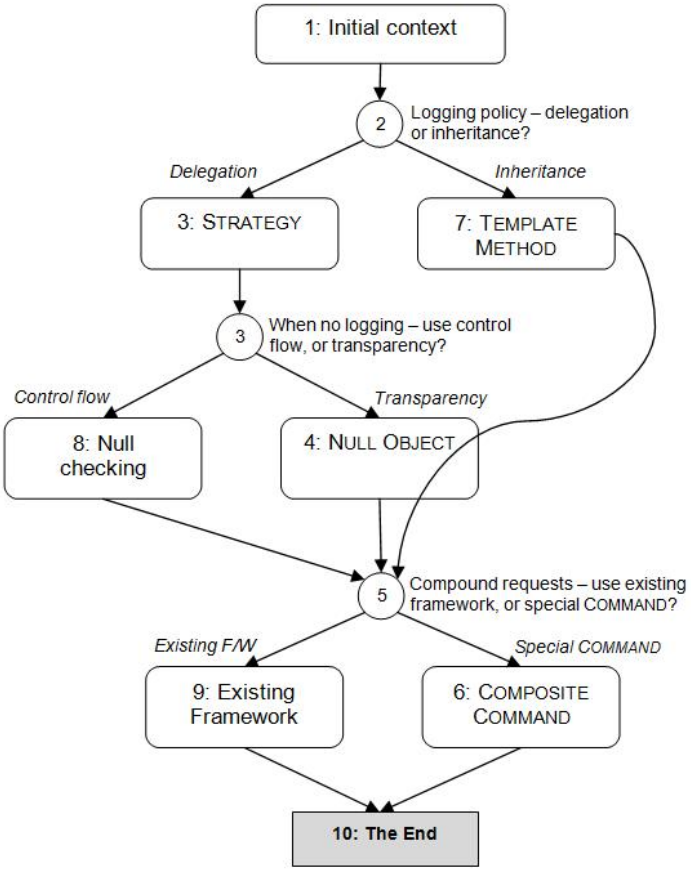

Fig. 3. Map of Story 1 - Varying Design Choices

4. Henney, K.: Context encapsulation. three stories, a language, and some sequences. EuroPLoP Proceedings (2005)
5. Montfort, N.: Twisty Little Passages: An Approach to Interactive Fiction. MIT Press, Cambridge, MA, USA (2004)
6. Buschmann, F., Henney, K., Schmidt, D.C.: 7. Pattern Sequences. In: Pattern-Oriented Software Architecture Volume 5: On Patterns and Pattern Languages. John Wiley and Sons (2007) 186–188
7. Buschmann, F., Henney, K., Schmidt, D.C.: Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing. John Wiley and Sons (2007)
8. Buschmann, F., Henney, K., Schmidt, D.C.: 9. Elements of Language. In: Pattern-Oriented Software Architecture Volume 5: On Patterns and Pattern Languages. John Wiley and Sons (2007) 251–254
9. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice. Second edn. Addison Wesley (2003)
10. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns - Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
11. Buschmann, F., Henney, K., Schmidt, D.C.: Pattern-Oriented Software Architecture Volume 5: On Patterns and Pattern Languages. John Wiley and Sons (2007)
12. Martin, R.C., Riehle, D., Buschmann, F., eds.: Pattern languages of program design 3. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1997)