Design Reuse in Human-Computer Interaction and Software Engineering

Michael J. Mahemoff
B.Sc., B.Eng.(Hons.)

Department of Computer Science and Software Engineering
The University of Melbourne
Victoria, Australia

2001

Submitted in total fulfilment of the requirements of the degree of Doctor of Philosophy

Abstract

Practitioners of Human-Computer Interaction (HCI) have a wide range of techniques and methodologies at their disposal, but few avenues for reusing successful design solutions. A mature discipline of HCI requires a systematic approach to learning from past results. This thesis proposes a number of techniques for improving design reuse in HCI. Some apply specifically to high-level design while others also consider detailed software design.

The major focus is the adaptation of the pattern language concept which originated in building architecture and town planning. It is argued that the properties of pattern languages suit many concerns for HCI. This motivates an investigation concerning how pattern languages for HCI might be documented and used.

This thesis is especially concerned with highly-constrained pattern languages. Rather than producing a universal HCI pattern language, several languages are developed, each with a strong focus on a particular area. A tight scope, it is argued, provides a coherent language structure and therefore provides a strong degree of assistance to practitioners. Three such languages are documented: (a) The Safety-Usability Patterns language focuses on safety-critical systems and demonstrates how a pattern language can provide guidance for high-level design; (b) Multiple Model-View-Controller (MMVC) focuses on implementing a small set of tasks within the model-view-controller architecture, demonstrating how a pattern language can provide guidance on detailed software design and still address usability; (c) Planet focuses on internationalised systems and combines the previous approaches to mix high-level design guidance with detailed design guidance, demonstrating the interdisciplinary capability of pattern languages. Several other forms of reuse are also considered: (a) online repositories, which store knowledge about user characteristics; (b) generic tasks, which capture activities recurring across many applications; (c) reusable software components, which can be documented via design patterns.

Results of this work have been validated in several ways. An experiment with designers using generic tasks indicated that they are useful for rapid brainstorming of software functionality. A second experiment with designers using Safety-Usability patterns led to a set of guidelines for developers and pattern authors. The Online Repository concept has been demonstrated with a prototype website developed in conjunction with this project. Three programs have also been developed which exemplify most of the MMVC and Planet patterns. The results of this work are largely favourable for the techniques which have been

proposed. They indicate that pattern languages can facilitate reuse of high-level design, detailed software design, and a style of interdisciplinary design that bridges the various levels of abstraction which HCI must address.

Declaration

This is to certify that:

1. The thesis comprises only my original work except where indicated in the preface,
2. due acknowledgement has been made in the text to all other material used,
3. the thesis is less than 100,000 words in length, exclusive of tables, maps, bibliographies, and appendices.

Michael Mahemoff

Preface

Refereed Publications The following refereed publications were produced in conjunction with this thesis.

[1] M.J. Mahemoff and L.J. Johnston. Usability pattern languages: The "language" aspect. In M. Hirose, editor, *Human-Computer Interaction: Interact '97*, pages 350–358, Amsterdam, 2001. IOS Press (for IFIP).

[2] M.J. Mahemoff, A. Hussey, and L.J. Lorraine. Pattern-based reuse of successful design: Usability of safety-critical systems. In D.D. Grant and L. Sterling, editors, *Australian Software Engineering Conference (ASWEC) 2001*, pages 31–39. IEEE Computer Society, Los Alamitos, California, 2001.

[3] M. J. Mahemoff and L. J. Johnston. Brainstorming with generic tasks: An empirical investigation. In C. Paris, N. Ozkan, S. Howard, and S. Lu, editors, *OZCHI 2000 Proceedings*, pages 224–231. CHISIG, Ergonomics Society of Australia, Sydney, 2000.

[4] M. J. Mahemoff and L. J. Johnston. Reusing knowledge about users. In D. Zowghi, editor, *Fourth Australian Conference on Requirements Engineering (ACRE)*, pages 59–69. Macquarie University, Sydney, 1999.

[5] A. Hussey and M. J. Mahemoff. Safety Critical Usability: Pattern-based Reuse of Successful Design Concepts. In M. McNicol, editor, *4th Australian Workshop on Safety Critical Systems and Software*, pages 19–34. ACS, 1999.

[6] M. J. Mahemoff and L. J. Johnston. The Planet pattern language for software internationalisation. In *Pattern Languages of Programs 1999 Proceedings*, Monticello, IL, 1999. http://jerry.cs.uiuc.edu/~plop/plop99/proceedings/. Accessed September 5, 1999.

[7] M. J. Mahemoff. Incorporating usability in the software design process. In M.A. Sasse and C. Johnson, editors, *Human-Computer Interaction: Interact '97*, pages 686–687. IOS Press (for IFIP), Amsterdam, 1999.

[8] M. J. Mahemoff and L. J. Johnston. Software internationalisation: Implications for requirements engineering. In D. Fowler and L. Dawson, editors, *Third Australian Conference on Requirements Engineering (ACRE)*, pages 83–90. Deakin University, Geelong, Australia, 1998.

[9] M. J. Mahemoff and L. J. Johnston. Principles for a usability-oriented pattern language. In P. Calder and B. Thomas, editors, *OZCHI '98 Proceedings*, pages 132–139. IEEE Computer Society, Los Alamitos, CA, 1998.

[10] M. J. Mahemoff and L. J. Johnston. Pattern languages for usability: An investigation of alternative approaches. In J. Tanaka, editor, *Asia-Pacific Computer-Human Interaction (APCHI) '98 Proceedings*, pages 25–31. IEEE Computer Society, Los Alamitos, CA, 1998.

Unrefereed Publications The following unrefereed publications were produced in conjunction with this thesis.

[1] M.J. Mahemoff. Weaving high-level and low-level patterns: An extended version of the planet pattern language, 2001. Technical Report 2001/21, Computer Science and Software Engineering Department, The University of Melbourne. http://www.cs.mu.oz.au/ tr _submit/test/cover _db/mu _TR _2001 _21.html.

[2] M.J. Mahemoff and L.J. Johnston. A high-level pattern language for software internationalisation, 2001. Technical Report 2001/20, Computer Science and Software Engineering Department, The University of Melbourne. http://www.cs.mu.oz.au/tr _submit/test/cover _db/mu _TR _2001 _20.html.

[3] M.J. Mahemoff. A primer on usability patterns. *Simplicity (Newsletter of CHISIG, Ergonomics Society of Australia)*, 2001. To be published in May, 2001.

[4] M.J. Mahemoff and A. Hussey. Patterns for designing safety-critical interactive systems, 1999. Technical Report 1999/25, Computer Science and Software Engineering Department, The University of Melbourne. http://www.cs.mu.oz.au/tr _submit/test/cover _db/mu _TR _1999 _25.html.

[5] M.J. Mahemoff. Position statement for Interact 99 workshop on "Pattern languages: Creating a community", 1999.

[6] M. J. Mahemoff. Software engineering and human-computer interaction. In P. J. Benda, F. Vetere, and J. B. Fabre, editors, *Proceedings of the 1997 Australia-New Zealand Student Conference in Computer-Human Interaction*, pages 40–43. School of Information Technology, Swinburne University of Technology, Melbourne, Australia, 1997.

[7] M.J. Mahemoff. Position statement for Interact 97 workshop on "Integrating software engineering and HCI", 1997.

Acknowledgements

I would like to thank Lorraine Johnston, my primary supervisor who has offered useful direction, feedback, and support at every point of the thesis. I also thank Philip Dart, my other supervisor, for reviewing many plans and drafts in the latter part of my thesis. Other staff and students of the software engineering research group in the CSSE department, particularly Liz Hayward, Kevin Chan, and Paul Chesson, have provided useful feedback throughout the thesis. On the HCI side, I consider myself fortunate to have discussed my work on numerous occasions with staff and students at the Swinburne Computer-Human Interaction laboratory.

Several other people have had a significant impact on the content of this thesis. Andrew Hussey at the Software Verification Research Centre, University of Queensland, was my co-author for the Safety-Usability Patterns, and we contributed equally to this work. Andrew also reviewed some early model-view-controller research concerning the MARCO application. Todd Coram was an excellent shepherd for the high-level Planet patterns, and drove home the importance of writing generative, coherent, patterns. Jim Coplien also emphasised to me the importance of the language concept and provided useful feedback about the high-level Planet patterns.

I wish to thank the University of Melbourne CSSE Department for providing resources and allowing me to recruit students performing summer work as subjects in design experiments. I am grateful to the students themselves for these contributions, and my friends who looked at early plans for these studies. I appreciate the support provided by the Department of Education, Training, and Youth Affairs, in the form of an Australian Postgraduate Award, during my time as a full-time student. Finally, I thank my family and friends for providing support and encouragement over the past few years.

“Those who do not remember the past are condemned to repeat it”

George Santayana

Contents

List of Tables

List of Figures

Part I

Background

Chapter 1

Introduction

1.1 Motivation and Aims

Computers are used by almost every individual in a modern society — in overt forms such as word-processors and internet browsers, and in more subtle forms such as car controllers, mobile phones, and domestic appliances. Computers are also an integral component of many large-scale systems, such as stock exchanges and air traffic control systems, that are critical to the operations of a modern society. Just a few decades after the modern computer was created, we have become almost inseparable from digital technology. Notwithstanding the diversity of settings, there is a persistent element in almost all cases: the user population. Modern society is a complex network of human and computer agents operating together, and developers must ensure interaction between these agents is as smooth as possible. This thesis is concerned with the practice of designing for effective interaction between humans and computers.

The past decade has certainly seen improvements in human-computer interaction (HCI) design, evaluation, and development methodologies (e.g. [36, 172, 107]). However, there remains a signficant gap in the area of design reuse. Even when developers follow the more sophisticated analysis and design techniques, the chief avenue for improving their abilities is personal experience; it would be desirable if they could also learn via explicit advice. Advice may sometimes be offered in the form of principles and design rules, but these can often be vague and internally inconsistent. Having to reinvent the wheel in each new project is not only time-consuming; it means that system design is not leveraging effectively from previous observations, implying its quality will suffer accordingly. Learning from past experience is particularly important in HCI, since user reactions cannot be accurately predicted *a priori*.

A further issue is integration of HCI and software engineering (SE) practices. Even without a strong HCI concern, software is a complex product which consumes considerable resources to build, maintain, and enhance. A theory of interaction design will be signficantly more useful to practitioners if it is sympathetic to these issues. A number of design techniques and methodologies do aim to integrate HCI and SE (e.g.

[135, 195, 190]). These approaches certainly provide a well-structured process which can adequately address HCI and SE. However, they still require developers to apply techniques based on their own skills and experience.

There is an opportunity to accelerate the learning process by documenting explicitly knowledge acquired from previous work. Design reuse is a growing concern in software, resulting particularly from the rise of the design patterns paradigm [88]. If design reuse can apply to high-level HCI concepts, there is an opportunity to reuse knowledge about the mapping of HCI concepts to detailed software. The aims of this thesis therefore combine high-level reuse with detailed design reuse:

- To discover and evaluate techniques which facilitate reuse of existing *high-level* design features that have contributed to usability.
- To discover and evaluate techniques which facilitate reuse of existing *detailed* design features that have contributed to usability.
- To investigate how these techniques may be combined to improve their overall effectiveness. In particular, to consider whether it is possible to produce a knowledge base containing both high-level design knowledge and detailed design knowledge, such that the detailed design knowledge guides developers on the implementation of features suggested by the high-level design knowledge.

1.2 Design Reuse in HCI and SE

A key concept in this thesis is the design pattern approach, introduced in the field of building architecture and town-planning, and more recently popularised in software design. This thesis adapts the approach to fit the motivation of reuse in HCI. Three pattern languages have been created, each representing a different level of reuse, as depicted in Figure 1.1.

Safety-Usability Patterns are high-level patterns for design of safety-critical systems, patterns which cover concepts such as accurate execution of tasks and presentation of information. They show how high-level HCI design concepts can be encapsulated in a form which facilitates reuse.

The Multiple-Model View Controller (MMVC) language contains patterns for object-oriented design of systems built in the Model-View-Controller (MVC) [130] architectural style. They show how certain user tasks, such as sorting a list, are supported in MVC programs. Furthermore, they and provide guidance on how to design the system in a manner which will be suitable for end-users. The purpose is to show how even detailed software design patterns can have a positive impact on usability. Furthermore, the patterns can be related back to the high-level concept of user tasks, hence they demonstrate one way to reuse knowledge about the mapping from HCI design to detailed software design.

Planet takes this type of reuse a step further, since it is a pattern language for software internationalisation consisting of patterns with a range of abstraction levels. At its highest-level, there are organisation-

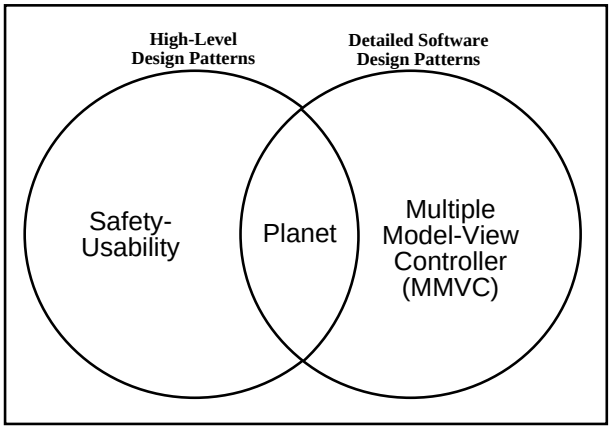


Figure 1.1. Pattern languages in this thesis: Safety-Usability Patterns, Planet Pattern Language for Software Internationsalition, Multiple Model-View-Controller (MMVC).

level patterns which deal with scheduling future versions and capturing knowledge about users from different cultures. From there, it maps into high-level design patterns relating to user-interface design, functionality, and configuration of preferences. These patterns are then related to several detailed design patterns, at a similar level to MMVC and conventional software design patterns.

The concept of design patterns is not the only form of reuse in this thesis. There are several other approaches which are related in some ways to the three pattern languages, but represent a different type of reuse (Figure 1.2). One approach is the online repository of knowledge about users, which is a form of high-level reuse which evolved from Planet. The idea is that knowledge about users captured in one project can be reused in future projects, hence an online repository is proposed to help an organisation or community maintain such knowledge. A prototype system, “Repository of Online Cultural Information” (ROCI), has been built to demonstrate the concept.

A further form of reuse is offered by the concept of generic tasks: task units such as “Compare” and “Add”, which recur across many applications. These can be used to provoke ideas which might not otherwise have been considered during requirements specification, so they may be used as either a brainstorming tool or an insurance mechanism. They also perform an additional role in this thesis, with a generic task list forming the basis of MMVC. MMVC shows how these generic tasks are implemented under an MVC architectural style.

MMVC itself is also more than just a pattern language. The MMVC patterns mentioned above are actually a subset of the MMVC framework. In addition to the patterns, there are reusable software components and a basic tool for source-code generation. The reusable components help designers implement MVC systems and support certain generic tasks. The source-code generation tool provides assistance to developers when creating typical classes for MVC. The patterns work with the components and tools, so

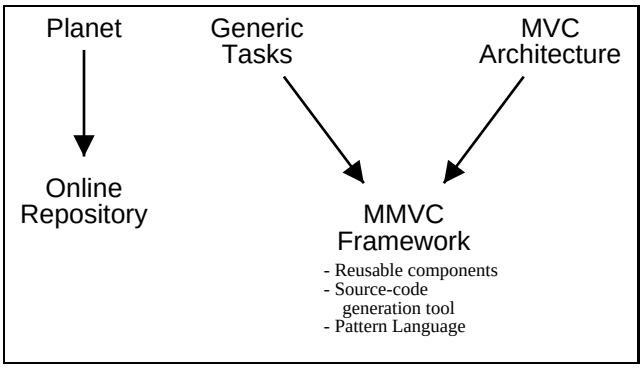


Figure 1.2. Additional reuse in this thesis, beyond pattern languages. Planet introduces the concept an online user repository. Generic tasks and the MMVC framework, of which patterns are only one aspect, are additional forms of reuse.

that the overall framework helps detailed software designers deliver usable software.

The work has been evaluated via experiments with designers as well as example software applications. One experiment involved asking designers to use the generic tasks as an aid to system design. Another experiment was concerned with usage of high-level design patterns, and considered how a pair of designers would use the patterns to approach a design problem. Two software applications were written in conjunction with MMVC. They informed its creation and also serve as examples. A further application, Critique, is partly based on MMVC, and therefore provides further validation of the approach. It has also been developed alongside the detailed patterns of Planet, and is used as an example in the patterns.

1.3 Conventions Used in this Thesis

Definitions and acronyms are described in Appendix H. One important clarification of terms must be mentioned here. In discussing design, there is an unfortunate ambiguity created by inconsistency between HCI and SE conventions. “Design” to an HCI specialist usually falls somewhere near the software engineer’s definition of “requirements”. To the software engineer, “Design” is a plan of the code for a target system, e.g. a statement of the system’s classes and their envisioned workings. Where there is a chance of ambiguity, I use “high-level design” to refer to the former type of design and “detailed software design” for the latter.

Throughout these pages, patterns are shown in `typewriter` font. Lines of code are also shown in this font.

Design diagrams, unless otherwise specified, are shown in Unified Modelling Language (UML) notation [84].

1.4 Thesis Overview

The thesis consists of four parts. The first part argues the case for design reuse in HCI and provides a background on pattern languages and recent approaches to patterns in HCI. The second part considers high-level design reuse. The third part extends this work by showing how high-level patterns can be integrated with patterns for detailed design reuse. The fourth part provides the conclusions of this work.

Stated in more detail, the remainder of **Part I** consists of the following chapters:

2. **Incorporating Human-Computer Interaction into the Software Engineering Process** Chapter 2 expands on the motivation presented here, detailing the tensions between HCI and SE and recent attempts to improve HCI practice. It motivates design reuse and explains why guidelines are an insufficient form of design reuse.

3. **Pattern Languages** Chapter 3 explains what patterns are and where they come from. This overview emphasises the importance of the "language" aspect of pattern languages, a theme which is carried through the entire thesis. The chapter finishes by presenting the argument for patterns in HCI.

4. **Applying Patterns to Human-Computer Interaction** Chapter 4 represents the transition from the abstract motivation of HCI patterns to a concrete view of this type of construct. It begins by presenting a classification scheme for HCI patterns. A large number of researchers have produced HCI patterns since the present work began, and this chapter shows where each approach fits into the classification scheme. It is observed that many approaches are quite broad in their scope, with only a few constrained to a particular domain or user type. Some of the more constrained collections exhibit a higher degree of cohesion among patterns, and the chapter concludes by showing how the three collections in this thesis fit into the classification scheme, showing that they also have a relatively tight scope. This chapter also covers related work other than pattern collections.

Part II consists of the following chapters:

5. **A High-level Pattern Language for Software Internationalisation** Chapter 5 introduces the high-level patterns of Planet, i.e. those concerned with modelling of cultures, system features, and user configuration of those features. It shows how Planet takes certain design principles and exposes the implications for developers in particular design contexts. This chapter also includes an overview of the software internationalisation domain which is Planet's central concern.

6. **Reusing Knowledge about Users** Chapter 6 expands on the online repository concepts introduced by Planet. The repository enables a group of developers to maintain an updatable store of knowledge about users. An envisionment scenario and design goals are presented. A working prototype of the system was developed in a project aligned to this thesis; a walkthrough is provided.

7. Safety-Critical Patterns: Theory and Empirical Study Chapter 7 introduces a second example of a high-level pattern language. It builds up a list of high-level design principles for usability in safety-critical systems, then introduces the Safety-Usability Patterns language which helps designers adhere to these principles. The patterns cover four areas of high-level design: inter-task management, task execution, presentation of information, and machine control. A controlled qualitative experiment was conducted, in which designers were asked to improve on an initial design sketch. Four pairs were given the Safety-Usability Patterns, four were provided only with Safety-Usability principles. The results led to suggestions for a number of practical guidelines for pattern authors and developers.

Part III consists of the following chapters:

8. Generic Tasks: Theory and Empirical Study Chapter 8 introduces the concept of generic tasks. It shows how a list of twenty-three generic tasks were captured from a range of software applications. A pilot study is then described in which six designers were asked to improve on two initial design sketches. A repeated-measures methodology was employed: the subjects initially used a standard structured technique, then revisited the design using generic tasks. The study provides evidence that generic tasks are an effective tool for rapid brainstorming at a requirements level.

9. Refining the Model-View-Controller (MVC) Architectural Pattern Chapter 9 does not deal specifically with usability. Instead, it sets the stage for the MMVC framework by introducing the MVC architectural pattern and exposing some of its difficulties. A sample application, MARCO illustrates MVC's limitations and shows how they can be dealt with.

10. MMVC: A Framework arising from the Implementation of Generic Tasks in MVC Chapter 10 combines the work in the previous two chapters to produce the MMVC framework. A sample application, Prescribe, is introduced, which supports the generic tasks under an MMVC framework. This chapter documents the reusable components, source-code generation tool, and patterns of MMVC. The framework is at the detailed design and coding level, but it allows work to be related back to user tasks, an HCI construct. Furthermore, principles of usability are encapsulated in the components and supporting patterns.

11. Weaving High-Level and Low-Level Patterns: An Extended Version of Planet Chapter 11 extends the high-level Planet patterns in Chapter 5. After showing how the detailed software patterns in MMVC relate to high-level tasks in Chapter 10, it is possible to investigate how detailed software design patterns can relate back not to tasks but to high-level patterns. The full version of Planet demonstrates that detailed patterns can indeed be closely associated with high-level patterns to provides a well-integrated pattern language. A sample application, Critique, was developed alongside the detailed patterns and exemplifies many of the patterns. The patterns also use other applications as further examples.

Part IV consists of the conclusions:

12. Conclusions Chapter 12 highlights the major theoretical contributions of the thesis. Future directions for design reuse in HCI and SE are also discussed.

Chapter 2

Incorporating Human-Computer Interaction into the Software Engineering Process

2.1 Introduction

This chapter outlines the current situation in Human-Computer Interaction (HCI) and its relationship to (SE). It looks at current approaches and explains why they do not cover knowledge reuse adequately. Section 2.2 explains that usability has traditionally been a secondary concern in software engineering and demonstrates why traditional software attributes conflict with usability attributes. In Section 2.3, existing approaches to the problem are compared — namely evaluation, development methodologies, and reuse. Section 2.4 discusses the extent to which reuse is currently possible, and highlights problems in this area. Section 2.5 summarises the situation and explains how this thesis aims to contribute to a better integration of HCI and SE.

2.2 Tensions between SE Practice and HCI Concerns

To appreciate the present place of HCI within the overall context of SE, a brief historical perspective is instructive. In a historical coverage of systems development, Friedman identified three types of constraints which have affected the field [85]:

Hardware Constraints Driven primarily by hardware costs and reliability.

Software Constraints Driven primarily by productivity of systems developers, difficulties of delivering reliable systems on time and within budget.

User Relations Constraints Driven primarily by system quality issues, including perception and catering for user needs.

According to Friedman, all of these constraints have applied at some level since modern computers were invented. However, he argues that the dominant constraint within systems development has shifted, from hardware to software to user relations. While the relative importance of these constraints will vary widely according to context, Friedman's constraints highlight the gradual rise of user needs as a primary consideration and indicate some of the difficulty the field of software engineering has in reconciling HCI concerns with those more entrenched issues of productivity and reliability. The increased importance of users occurred as interactive systems replaced the batch processing paradigm [97]. This, in turn, led to a much greater user base. No longer could developers assume they were creating products for their own kind. And as computers, in their various forms, are adopted not only by professionals but by almost everyone in modern society, developers of many systems cannot even assume there will be significant training or selective recruitment for many systems.

Other commentators have remarked on the dichotomy of perspectives, being either technology-focused or human-focused. Floyd has suggested there are two prevailing paradigms: product-oriented and process-oriented [81]. In the former perspective, software is viewed as a set of well-defined mathematical objects which can be explicitly specified, and specified upfront. The latter perspective considers software as "tools or working environments for people", in which development proceeds iteratively.

Allan Cooper, an HCI author with considerable practical experience, argues that the psychology of the *programmer* is necessarily at odds with the goals of user-centered design [51]. Programmers, in his experience, tend to favour control over simplicity, present users with models which reflect implementation details, focus too heavily on unusual cases instead of streamlining typical situations, and can be disparaging towards users. Although the last observation may not be the case with a quality-oriented software engineer, the fact remains that even well-meaning software engineers will experience some tension in simultaneously focusing on the user's needs as well as the underlying technology.

The tensions between software engineering and HCI can be summarised by considering the quality attributes which characterise a software project, and asking how they relate to each other. In some sense, this is an expansion of Friedman's work on the evolution of constraints. Some important software attributes are:

Efficiency Amount of resources used by the system (e.g. memory, processor time).

Reliability Degree to which program conforms to expected functionality.

Testability Amount of effort required for testing.

Maintainability Amount of effort required to correct system deficiencies.

Portability Amount of effort required to introduce the program to a new platform.

These factors can conflict with each other. For example, a system designed for portability is likely to be less flexible, since any new function must work across multiple platforms. A reliable system may contain redundant code for the purposes of preventing or trapping errors; it is likely to be less efficient. Developers must apply various engineering techniques to prioritise among the various attributes and create their systems in ways which resolve these conflicts.

It is somewhat misleading to refer to usability as "yet another software attribute". When usability becomes a consideration, as it should for any interactive program, the situation becomes substantially more complex. As the above discussion suggests, the usability factor represents a significant shift in thought. The system is no longer a deterministic machine, but rather a machine interacting with a heterogeneous set of operators whose behaviour may be difficult to predict. Following is a discussion of the tensions caused by this situation, with respect to the software attributes mentioned previously:

Usability versus Efficiency Features of software which improve usability consume computational resources. Most notably, running the user-interface of a typical desktop application (e.g. a word-processor) often consumes many more resources than performing activities such as arithmetic calculations or variable manipulation. There are also other features which consume more resources than would be necessary if usability was a low consideration. For instance, it is possible to improve flexibility by supporting user preferences, but this functionality consumes both time and memory.

Usability versus Reliability To ensure a system is reliable, it must take into account the range of possible user actions; unanticipated user activity will be likely to cause some undesired system actions. In a non-interactive program, this is quite easy — a certain range of inputs can be expected and the program can be engineered to deal with these inputs. However, an interactive program must anticipate the behaviour of a variety of human users, some of whom may have limited computer experience and therefore exhibit particularly surprising behaviour. What if an invalid date is entered? What if the user tries to delete important data? Allowing people to use software freely while maintaining integrity can be a challenging task.

Usability versus Testability Evaluating an interactive system can be done in three ways: observing a representative end-user interact with the system, recruiting a specialist to assess the system, and performing automated testing. The first technique leads to numerous problems, such as unavailability of users, under-representation of real-life scenarios and environments (see Section 2.3.1). The second technique denies the fact that the testing professional does not have the same perspective as the user, and cannot possibly anticipate all actions a user might perform in a real-life situation. Automated testing can help when data sources are highly-constrained, but this will not be the case if the software is flexible to users' needs. Furthermore, GUIs significantly complicate the process of automated testing. A non-interactive test script could simply call a function with various values, while a GUI tester must use a tool to simulate precise mouse movements and button-clicks, or perform some type

of complicated inter-process communication with the application.

Usability versus Maintainability A well-accepted phenomenon in sofware engineering is that maintainability suffers when changes are made too frequently and without appropriate reflection. Yet, a frequently-cited goal in HCI is to incorporate results from evaluation, and this provokes many HCI specialists to promote iterative development cycles [216]. Iteration is certainly a reasonable way to accommodate user feedback. However, software can sometimes be deceptively easy to alter. With hardware, a re-engineering effort might take weeks or months, and would therefore be more likely to necessitate a period of serious reflection prior to any work taking place. With software, a change of the same impact to end-users might require a few hours by a single programmer. However, the ease belies the underlying risks of altering a complex piece of software. The programmer may not be aware of certain assumptions which have been made by other programmers or software libraries, and such problems can remain undetected until the moment a user performs some unanticipated action. Quick, unconsidered, changes during iterative development can cause the system to degrade over time, in contrast to software produced under a more controlled lifecycle model, such as the classic waterfall model [189] or spiral model [22].

Usability versus Portability Hardware/OS platforms tend to promote certain conventions in order to provide a consistent look-and-feel to the user (the best known is Apple's Human Interface Guidelines [7]). This is a valid approach to usability; a successful transfer should respect the "local customs" of the target operating system if usability is to be maintained. The Quicktime player, ported from Apple to Windows, demonstrates what can go wrong; it contains no menubar to help the user move or maximise, and is rendered in an unalterable Apple-like colour scheme [8]. Adapting to new features increases the overall usability effort. Furthermore, GUIs in general impose significantly more portability complications, typically requiring a cross-platform toolkit, or a refactoring of the application. Often, developers resort to a "lowest-common denominator" approach, i.e. use only those services which are provided by all target platforms. This restriction decreases the quality of user experience and reduces consistency with other applications on the same platform.

The tension between usability and other attributes is symptomatic of a larger problem: the tendency for members of the HCI and SE communities to view HCI and SE as strictly opposing paradigms, and often to ignore the other side's argument. HCI people sometimes argue for approaches such as iterative development and user-centric features without acknowledging the technical arguments against these approaches. Often, these technical reasons are compelling, as discussed above. On the other hand, software engineers, as Cooper pointed out [51], often ignore users, or impose their own, heavily-skewed, traits onto their model of the user.

The present thesis takes the perspective that progress will be made once both sides are understood. It was mentioned earlier that there is even tension among the non-usability attributes. Many researchers in

computer science and software engineering therefore seek to find optimal solutions. Sorting algorithms, for example, represent attempts to optimise factors such as speed, storage capacity, and algorithmic complexity. In the same way, this thesis is concerned with the forces which software and HCI practitioners respectively encounter. I have strived to address the needs of both sides in their own terms. The HCI considerations in the thesis are high-level and deal with user experiences rather than such technical abstractions as "degree of reversibility" or "number of mouse operations to access function X", which would have less relevance to the goal of usability. The software considerations, in contrast, are detailed down to the level of code. For example, the thesis will explain how certain features of architectures or code can affect properties of the software associated with usability. While it is impossible to address all possible HCI or software forces, the thesis describes approaches which can enable software development to progress with first-class consideration of HCI and software issues.

2.3 Incorporating HCI into the Software Process: Major Approaches

There are several approaches which can help to resolve the tensions between software engineering and HCI. In many cases, they are complementary to each other. This section examines the major approaches.

2.3.1 Evaluation via Testing and Expert Assessment

Evaluation is a natural process whenever a product is to be released to a user group. It is therefore not surprising that methodologies for evaluation of software have evolved over many years, and this includes many approaches to usability evaluation of software.

2.3.1.1 Usability Testing

In usability tests, a user is observed interacting with a running product. The product may vary in state from early prototype to a completed product. Data may be drawn from questionnaires, software logs, observations by an administrator, and interviews with an administrator [199]. Even experienced HCI practitioners can fail in their attempts to predict user behaviour; testing with representative subjects helps to ensure at least a subset of defects are found before the software is deployed for real-world usage.

Nevertheless, usability tests do impose many difficulties. A foremost concern is user availability. Representative users cost money, especially when the target audience consists of people in specialised roles. A test with few users will have lower user diversity and will not enable quantitative claims, a situation which reduces the test's validity. Even when many users are available, their time will usually be limited to a few hours. Two detrimental results follow: (a) only a few aspects of the software can be considered, and (b) instead of demonstrating the experience of users with varied levels of experience, the test is typically focused on a novice user learning the system [204].

There are additional overheads. The test must be devised in a scientific manner, ensuring that results will be meaningful. As opposed to the automated statistical testing often performed by computer scientists, usability testing requires methodologies which enable inferences to be drawn about subjects' mental processing. This is an exercise which requires not only an appreciation of the software project and its objectives, but also knowledge of human psychology.

Recruiting subjects can be a lengthy exercise in itself, with subjects acquired from sources such as clients, employment agencies, internal personnel departments, universities, or social contacts [199]. Ethics approval may also be required, thus assessors must design experiments with minimal risk to subjects, and spend further time justifying the risk.

2.3.1.2 Expert Evaluation

One way to lessen the difficulties of user testing is to combine it with expert evaluation. Typically, experts spend a few hours interacting with the application, recording as many usability defects as possible. Shneiderman lists five forms of evaluation [204]:

Heuristic Evaluation Experts critique according to a list of heuristics such as "Provide timely feedback".

Guidelines Review Experts check the interface against a list of detailed guidelines, such as "Use a progress bar for activities which take more than 5 seconds".

Consistency Inspection Experts check for consistency across the application.

Cognitive Walkthrough Experts simulate a user conducting various tasks.

Formal Usability Inspection A moderator presents the application and experts analyse it.

Expert evaluations reduce the resourcing complications of testing with end-users. Expert evaluators are still a required resource, although there are techniques to reduce this dependence. For instance, Nielsen argues that his 10 heuristics catch a very large proportion of the problems which are supposed to be caught by collections of 100-1000 design rules [172].

However, expert evaluations alone are not satisfactory. User tests will often reveal errors unanticipated by evaluators, and there is also a risk evaluators will overestimate the significance of minor defects. In a comparative study, Karat et al. [124] found user testing yielded more than twice the quantity of problems that expert evaluations delivered. Furthermore, they noted that the user test results were accompanied by better contextual information. They also observed evaluators but being so distracted by cognitive walkthroughs they were performing that they ommitted to record some defects they had observed.

It must also be remembered that an expert in software usability is often not an expert in the underlying domain, such as medicine or financial systems [197]. Clearly, domain expertise is also required to capture many of the problems which could have severe implications in a production system.

A problem shared by expert evaluation and user testing is poor timeliness. Evaluation often takes place after development, when it is too late to be useful. This practice may suggest a few quick fixes, but is unlikely to provoke substantial change. This is unfortunate, because many usability defects cannot be solved by a quick fix. They relate to human-computer dialogue and task context, and may necessitate significant changes to the software architecture. This kind of change is rarely practical at the tail end of a project; the expense may be excessive, developers may no longer be available, the window of opportunity for the application may be approaching. The psychology of the coders may further hamper the situation; even if they are motivated to re-engineer the system, it is not unusual for a degree of functional fixedness to restrict the extent they are able to re-design.

To combat the late evaluation problem, it is possible to perform some expert evaluation at earlier stages. Experts can look for defects in such artifacts as early prototypes and user-interface sketches. Users can be consulted to evaluate paper or software prototypes. Task analysis techniques such as the GOMS family [121] can also inspire an understanding of the system before it is coded.

Regardless of how or when it occurs, evaluation has one fundamental drawback: it is concerned with identifying problems rather than fixing them. The tendency to over-focus on evaluation has earned some practitioners the dubious title of "the usability police" from coders who feel that HCI practitioners criticise existing work rather than offer new ideas. This is too harsh a view for a practice which is certainly desirable on most projects. Nevertheless, it does reflect the need for complementary techniques which fix usability defects and, moreover, prevent them in the first place.

2.3.2 Lifecycle Models

In software engineering, the archetypal lifecycle model is the linear sequential model (based on Royce's waterfall model [198][1]). The model seems to be the oldest and most widely used [189]. Essentially, the model advocates several stages in direct sequence, typically along the lines of: concept generation, analysis, design, code, testing, maintenance (Figure 2.1). Although there are some variations, such as the ability to retreat one stage, the overall philosophy is to plan ahead carefully, and constantly progress through the stages. The chief motivation for this approach is that each activity will be more effective and yield more valid results if care is taken beforehand.

While the linear sequential model does have theoretical benefits, it has often failed to live up to expectations in real-world projects. In particular, HCI researchers since the 1980s have paid considerable attention to improving the situation [98]. From an HCI perspective, the main complaint is that a linear lifecycle does not enable sufficient feedback from users. While it is possible to consult users at the requirements stage, their views are likely to change once they see a user-interface design. And once users are interacting with a fully-coded system, further change requests are likely to emerge.

[1] Royce's original waterfall included feedback loops, but Pressman [189] notes that most organisations applying the model do so in a strictly linear fashion.

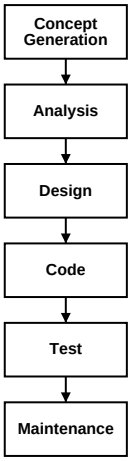


Figure 2.1. Linear sequential lifecycle model. Actual activities may vary across projects and organisations (based on Royce, 1970 [198]; Pressman, 1997 [189]).

A key complaint with lifecycle models, then, is integrating HCI activities with SE activities. SE lifecycle models such as the linear sequential model neglect HCI activities. Similarly, structured approaches to HCI activities do not necessarily address SE considerations. Nielsen's usability engineering lifecycle [172] is well-known in HCI, yet it addresses only HCI activities — studying the user, prototyping, evaluation, etc. It is useful as a statement of best practice in HCI, but a developer is left with the task of combining this lifecycle with the SE lifecycle. Since this is likely to be a complex task, it would seem useful to produce a single lifecycle model taking into account SE activities, HCI activities, and the interaction between them.

Indeed, a plethora of lifecycle models have been proposed to reconcile HCI and SE perspectives. One prominent proposal for iterative development with early user involvement dating was issued by Gould and Lewis at the first CHI conference, in 1983 [91]. In HCI circles, perhaps the two best known responses to the linear sequential model are: (a) Boehm's spiral model [22], a model which arose from an SE perspective, and (b) Hix and Hartson's star lifecycle model [107], more closely associated with the HCI discipline.

The spiral model adopts risk management as its central principle (Figure 2.2). The model encourages developers to cycle through various stages, while providing enough decision-making structure to ensure each iteration has appropriate objectives. A cycle is comprised of four stages. The first involves deciding on cycle objectives, planning the cycle, and identifying options. A risk analysis is conducted in the second stage, and the options are compared to produce a detailed understanding of the key activities for this cycle. The key activities themselves, such as requirements elicitation or design or coding, occur in the third stage of each cycle. Finally, the work performed in cycle is evaluated, the overall plan is considered in light of the past cycle. The developers then decide whether it is worthwhile continuing the project, and if so, future plans are made.

From an HCI perspective, the spiral model is more amenable to prototyping than the waterfall model.

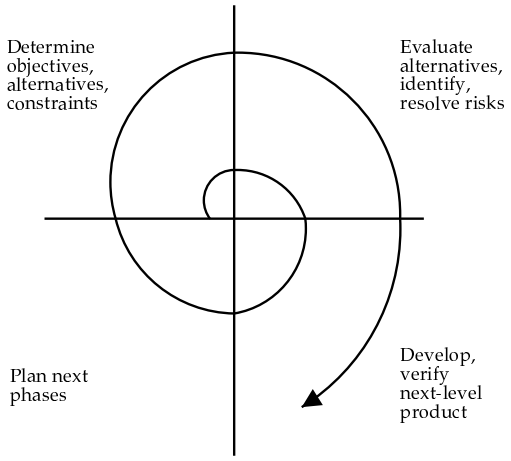


Figure 2.2. Boehm's spiral lifecycle model (based on Boehm, 1988 [22]).

If a manager decides that user requirements are not well enough understood, and therefore pose a risk to the application's usability objectives being met, the manager may decide the cycle will be used for prototyping. A more recent version of the lifecycle model is the win-win spiral model [23], which includes a more thorough consideration of all stakeholders' objectives. This is a further improvement over the waterfall model, since it allows for a more explicit consideration of user needs.

However, the Spiral lifecycle model still does not prevent confusion surrounding the integration of HCI processes. For instance, does user-interface design occur during requirements specification or software design? This lack of direction can lead to poor usability, a view which is supported empirically [167]. HCI consideration is essential to modern projects, hence it is reasonable that lifecycle models of modern projects explicitly acknowledge HCI activity.

Hix and Hartson explained that their model resembles Boehm's spiral, but is "specifically for improving usability of the interface" (p.485) [103]. Where risk management is the basis of the spiral model, usability evaluation is fundamental to the star lifecycle (Figure 2.3). The important property of this lifecycle is that a usability evaluation must take place between any two phases. Developers are free to switch from any phase to any other phase (or immediately back to the same phase), as long as this evaluation occurs in the transition. The nature of evaluation is naturally tied to the preceding activity; it could vary from a requirements traceability analysis to a software usability test. The model enables developers to shift between concrete examples and high-level models, and clearly encourages prototyping.

An appropriate lifecycle model can legitimise those processes which are necessary for usability consideration, such as prototyping and user testing. With the HCI lifecycles, iteration is an essential component. A lifecycle model in itself does not ensure correct techniques are followed, nor does it ensure techniques are followed correctly. It simply provides a useful framework around which a project may be structured.

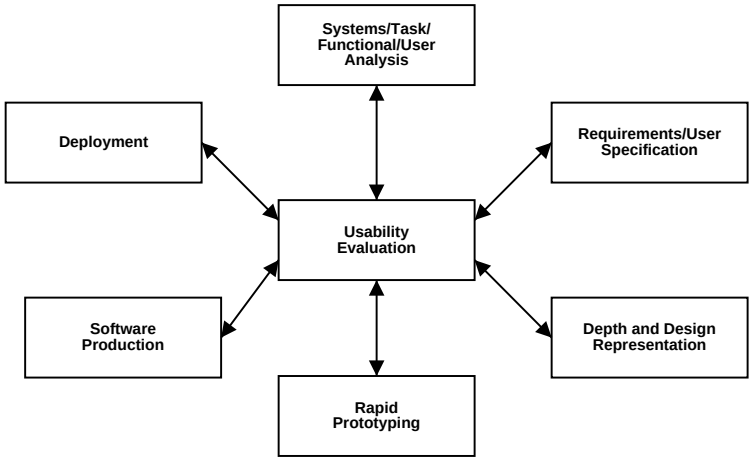


Figure 2.3. Hix and Hartson's star lifecycle model (based on Hix and Hartson, 1993 [107]).

2.3.3 Development Methodologies

Lifecycle models like the spiral and star models are quite broad. They deliberately avoid commitment to particular techniques. The resulting generality is a boon which enables the models to be adopted in a diverse range of projects. Organisations can tailor the models if they need, adjusting the overall flow as well as choosing specific activities to meet the objectives of each stage.

However, the refining process comes at a cost. First, there is the obvious effort of deciding how best to tailor the process. To be effective, this will require management support as well as a large inter-disciplinary effort across the organisation. More concerning than the effort is the question of quality. By electing to develop only from a skeleton of a lifecycle, the organisation is taking a gamble that they can create an adequate process internally.

So what is the alternative? One alternative has been present in SE circles for many years: an integrated development methodology. A coherent set of techniques is proposed, and these techniques may be applied across a corresponding lifecycle, a lifecycle which is also part of the methodology. An early SE example is the JSD/JSP approaches [118] (see below).

In SE, the most commonly practiced methodology at this time is probably the Rational Unified Process [84]. The lifecycle begins with an Inception Phase, where the project is approved. In the Elaboration Phase, requirements are fully specified. Continuing the recent trend of SE models allowing for iteration, the Construction Phase consists of any number of cycles, with activities including design, coding, and evaluation. The project is finalised in the Transition Phase, incorporating activities like beta testing and user training.

While this approach is somewhat less comprehensive than many corresponding academic approaches,

the process is well-known and contains some important observations to be drawn about coherence. Use cases are developed during the elaboration stage. These constructs enable the designer to specify, at a general level, typical usage scenarios which will arise during the system's operation. When construction begins, developers study use cases and usage scenarios, and derive classes by considering the entities involved during usage. By analysing the messages sent between these entities, class methods are also obtained. As this refinement occurs, a class diagram can be drawn to capture the relationships among classes. Complex classes can later be designed more clearly with the aid of state diagrams. In a later phase, these design artifacts will form the basis of coding activity. Later still, testers will compare the running product against use cases, checking for conformance.

The Rational Unified Process demonstrates how a series of techniques and artifacts can be combined to form a development process which is coherent. The activities are designed to inter-weave with each other, the lifecycle model ensuring they work together effectively. An organisation which adopts a methodology like this can benefit from the experiences which have been drawn upon to derive the methodology, even if it chooses to tailor the methodology somewhat. When the methodology is enhanced, the organisation inherits the benefits. When tools are released, the organisation can utilise their full potential.

Explicit development methodologies for HCI help to bring it closer to being an engineering discipline as opposed to a craft, a move which has been encouraged by a number of practitioners [162]. An engineering discipline, because it relies on explicitly-stated techniques, can aid practitioners and make their work practice more transparent. Furthermore, documenting state-of-the-art approaches enables gradual improvement by the community. This could be done with individual techniques, but is more easily studied in the context of methodologies which take full, or at least dominant, responsibility for project execution.

A practical methodology for HCI must deal not only with user-centered activities, but also with their relationship to software development activities. These attributes are embodied in Lim and Long's MUSE ("Method for USability Engineering") [135], a well-known tool in HCI circles and one which has been applied in a number of real-life projects [72]. MUSE paves a parallel path for the SE and HCI activities with "handshake points" along the way. The HCI side contains three phases, each composed of several stages. The first produces a general model of the tasks which are performed on "extant" systems (related, existing, systems). The second phase (among other things) decomposes the general model to obtain a model of the system tasks and another of the user tasks. The third phase produces interface designs. At this point, HCI and SE groups team up to follow a joint implementation and evaluation.

While this lifecycle model is useful in its own right, MUSE derives further power from its creators' decision to use the JSD* notation. This notation is derived from, and still closely related to, Jackson System Development [118], a well-known SE design technique incorporating a notation for temporal models of system activity. In MUSE, JSD* models provides a common basis for various HCI activities. In the first phase, practitioners begin by creating JSD* diagrams of extant systems. A generalised model of these extant systems is then produced by synthesising the various JSD models, an example of the smooth

interaction between the different stages. A target task model is also produced. In the next phase, a statement of user needs is generated, outlining human factors or operational concerns such as domain details and performance requirements. In light of this statement, the subsequent stage synthesises the two generalised task models. The process continues on to joint implementation and evaluation.

Several points should be noted about MUSE. First, the stages are centered around one particular modelling tool, JSD*, and explain how JSD* models can be manipulated. This way, a practitioner can perform the appropriate manipulation — such as synthesis or decomposition — whenever the MUSE lifecycle dictates it is necessary. Second, the model prescribes an approach to co-ordinating SE and HCI. For example, the outcome of the statement of user needs and composite task model is fed into the SE effort to produce a list of functions. Furthermore, the JSD* models themselves fit nicely with SE design, given that they are closely related to the software engineering technique, JSD. For all its attention to detail, MUSE is still a very flexible process; each stage can proceed using a variety of existing techniques. In fact, the authors outline the alternatives at each stage.

Several other methodologies work at a similar level to MUSE. OVID ("Object, View, and Interaction Design") is an iterative method incorporating HCI into an object-oriented development methodology [195]. Objects are identified on the class diagram, views of the classes are attached on the class diagram, tasks are specified using sequence diagrams, and interactions are expressed via state diagrams for particular classes. OVID uses object-oriented expression as a common reference point among activities, be they predominately HCI or predominately SE. Among other well-known methodologies is Hix and Hartson's user-centered methodology [107], which expands on the star lifecycle to provide more prescriptive information about working within and between stages. Constantine and Lockwood's Usage-Centered Design [48] is a cohesive approach in which most activities relate to Essential Use Cases, constructs similar to Rational's Use Cases, but with an intent to be extremely simple, general, and technology-independent.

Cohesive development methodologies can provide organisations with a well-considered framework which nevertheless retains a good degree of flexibility. Developers can benefit from enhancements, as well as by-products such as tools, training, and some familiarity among consultants and potential employees. They are a form of *process* reuse, since their creators can observe how practioners use them and continually improve them. However, they generally do not allow for ongoing *knowledge* reuse.

2.3.4 Software Support

Development methodologies and individual techniques can be augmented with supporting software. This software may be a tool which facilitates a development activity, such as a code-generation tool or a drawing tool. It may be capable of assessing a proposed design. Or it might perform an activity while the user executes the program.

As with development methodologies, there are numerous SE precursors. In the case of Rational's Unified Process highlighted in the previous section, Rational provides the comprehensive tool suite Rational

Rose [192]. It enables visual modelling of the various Unified Modelling Language (UML) constructs, with cross-referencing between models. Design models can be converted into modifiable code, and modifiable code can be reverse-engineered into design models. Peripheral activities can also be incorporated, such as version control, document generation, and project management.

Several model-based interface approaches have been developed to aid HCI development [190, 137, 21, 73, 229]. Typically, a designer works with an interactive tool to specify information such as domain specifics, desired tasks, and user characteristics. Often, the software is capable of generating a specification of some aspect of the final system, or a prototype of the final system, or may be able to critique the model based on human factors knowledge.

MOBI-D [190] is a good example, because it is comprehensive and relatively well-known. It consists of an architecture with well-defined specification syntax, a set of supporting tools, and surrounding lifecycle model. The process is centered around an interface model and a repository which contains information such as user-interface components, domain objects, user types. Tool support is offered at a range of levels. Tools can generate interface designs, with automated layout if desired. Knowledge bases can be invoked to guide generation of artifacts or assess quality of artifacts. Runtime tools can generate online help and capture user interactions.

Luo, a developer of the MIDAS environment, provides a reminder that software tools should be developed along HCI principles, just as any other software should be [137]. The philosophy behind MIDAS (similar to other products) is that the computer performs those tasks which are cumbersome for the designer (e.g. detecting interactions among design decisions and manipulating interface models), and allows the designer to practice what they do best. A primary goal in any software is to allow the user to perform their work transparently, i.e. focus on their work-related tasks rather than the system they are using. Tools such as MIDAS work toward this goal by automating those design tasks which are tedious and capable of being performed by computers.

Initially, these systems often focused too much on the domain model at the expense of user tasks. For instance, a window may be drawn for each object in the system, ignoring the ways in which users would need to use the application. Approaches like MOBI-D capture enough information to overcome this problem. Information relating to user characteristics is also being incorporated into these approaches.

The methodologies mentioned earlier, such as MUSE and OVID, could provide greater tool support but their developers apparently prefer to focus on other areas. Whether tool support is provided or not, the opportunity for knowledge reuse nevertheless remains. There is likely to be much commonality across projects. MUSE comes close to this approach with its extant systems analysis — suggesting for instance, that the manufacturer of a ticket booking system consider an existing related system. The situation could be further improved if people began to capture and document generalised models of extant systems. This would accelerate the development of extant system models for new projects. In fact, it is feasible to catalogue generalised versions of the various artifacts used in the methodologies discussed here. These

include task models, user-interface designs, and user models.

2.4 Knowledge Reuse as a means of Improving SE-HCI Integration

Each of the above approaches to SE-HCI integration has its merits. Certainly, these approaches can be combined to produce a mature approach to usability. This thesis considers a further step forward: reuse of existing knowledge. Brooks has explained that a characteristic of engineering science is that practitioners tend *not* to derive knowledge from basic science, even when it is feasible to do so [32]. Brooks noted that civil engineers can learn generalised construction techniques, such as thin core construction or homogeneous construction, which accelerates their design process. Their work is further accelerated by studying categories of problem, e.g. foundation-soil or earthquake hazard.

HCI practitioners can benefit by labelling patterns of problems such as *User adds item to work material*, which could relate as much to a composer dragging a musical note into a ledger as to a manager incorporating a new staff member into a project plan. These generalised problems may lead to one or more generalised solutions, e.g. *Show amount of elements in work material*.

It may be argued that reuse in HCI is even more important than in other disciplines, because HCI practitioners often *cannot* derive knowledge from basic science. The basic sciences in HCI are computer science and psychology. The behaviour of computers is well-understood and computer scientists can certainly derive observations from the basic science if necessary. Human psychology is less predictable, to say the least. To validate this claim, one only has to observe the arguments throughout the past century on the fundamental issues of human behaviour and thought, such as whether humans should be modelled via Freudian notions of the conscience, via behaviourism, via information-processing theory, or via neural networks. This may be contrasted to the operation of computers, where every CPU cycle, memory transfer, and output process is essentially deterministic and verifiable.

The net effect is that HCI practitioners desiring to improve their skills have little choice but to learn from past observations. Rather than expecting HCI practitioners to learn in isolation, the community as a whole must find ways to collectively capture information and disseminate it. Such mechanisms for reuse are a primary motivation of this thesis.

There is already a degree of reuse in HCI. The approaches to HCI-SE integration mentioned earlier are themselves reusable. The developers of MUSE, for instance, have devoted considerable effort to developing their methodology and continue to improve it based on observations. In this sense, a developer adopting MUSE is leveraging from existing experience. Likewise for someone who uses MOBI-D and its software tools. However, as already mentioned, the knowledge base is not inherited. Ten developers who use MOBI-D will go through many of the same learning processes, each in relative isolation from the others. The developers may share knowledge informally at a conference or on a mailing list, but it would be more efficient if there were mechanisms designed specifically for documenting the patterns of artifacts which

arise in using such a methodology.

As well as reusing techniques, developers can sometimes reuse or extend code. This form of reuse has been of great assistance in the past two decades. No longer do developers have to perform low-level tasks such as manually drawing widgets and using a mathematical formula involving screen co-ordinates to determine which widget the mouse resides in. Graphical operating systems and software libraries such as Smalltalk's Model-View-Controller framework have made user-interface design a much simpler and less tedious task. However, usability comes from a consideration of the users, their usage context, and the tasks they wish to achieve. These high-level concepts could not reasonably be encapsulated in software code, hence a higher-level form of reuse is required.

In fact, there is a high-level form of reuse which has been already available for many years: design guidelines. Guidelines include high-level principles such as *Feedback and Dialog* [7]. They also include more detailed assertion's like Smith and Mosier's *Easy Cursor Movement to Data Fields* [205]. Both varieties have their weaknesses. Broad design principles are often little more than vague truisms — it is difficult to deny the value of *Feedback and Dialog*. On the other hand, what does it mean to a designer with a real design problem at hand? How much information should be shown after an action takes place, and does it depend on the importance of the action? High-level principles also conflict, a further complication. It may be necessary to provide user feedback in substantial detail, but a developer could struggle to reconcile this advice with a principle of aesthetic integrity which advises against cluttering the user-interface.

Low-level design rules can also be flawed. Firstly, there are often many rules to adhere to; Smith and Mosier's classic set of guidelines [205] contains 944 guidelines, leading to critical indexing and retrieval issues. The low-level detail means that many design rules must be specific to a particular platform or organisation. As with principles, situations will arise when rules conflict with each other, and this may even be difficult to detect with so many rules to follow. Rules may also fail to adequately take into account user needs which may be established by task analyses and related techniques.

Criticism of design guidelines is not limited to speculation. There is empirical evidence in the form of studies with designers using sets of guidelines. De Souza and Bevan considered drafts for ISO menu design standards [68]. Three designers found that 91% of the guidelines led to difficulties for at least one designer. Difficulties included lack of clarity, repetition, and vague guidelines scope. Tetzlaff and Schwartz found designers only skimmed guidelines, and suggested guidelines should be used primarily to complement toolkits and interactive examples [217].

2.5 Discussion

There are fundamental tensions between SE and HCI. SE has historically focused on "hard" attributes like efficiency and reliability. Later on, developer resources also became a major issue, leading to higher prioritisation for attributes such as testability, flexibility, and portability. Only in recent times has usability

become a fundamental issue for system development. This historical progression exacerbates the basic tensions between the disciplines.

There have been many approaches to improve the quality of HCI work. These approaches have their merits and can be successfully combined together. Usability evaluation is essential, but it does not improve quality of applications in the first place. Lifecycle models help to structure the process, but leave open the activities performed at each stage. There are more specific methodologies which do give detail at each stage, such as MUSE and OVID, as well as methodologies like MOBI-D which focus more on tool support.

In all of these approaches, there are opportunities for reuse of existing knowledge. As already discussed, the major forms of reuse are principles and reusable code bases. Both types of reuse have significant problems, and it seems reasonable to ask what other ways reuse could occur. This thesis proposes further means of knowledge reuse, and studies effectiveness of these proposals.

The nature of knowledge which may be reused varies, but a common theme is that reuse of the knowledge should facilitate improved consideration of usability. Reuse of this nature can involve reuse of existing high-level designs (e.g. user-interface specifications), detailed software designs (e.g. class relationships), and software code. It can also imply reuse of knowledge about users and their tasks.

The thesis treats both HCI and SE as first-class concerns. It considers key HCI concepts such as user needs, tasks, and usage context. Likewise, typical software design and coding concepts are also considered. Accepted principles of design in both HCI and SE are adhered to.

In this chapter, knowledge reuse has been motivated in general terms. But the notion of reuse must be refined. Are we discussing reuse by an organisation or reuse by the entire community? What exactly is being reused? How does the reuse occur? The next two chapters look at the concept of pattern languages, discussing how they can facilitate reuse.

Chapter 3

Pattern Languages

3.1 Introduction

In the previous chapter, it was argued that knowledge reuse would improve integration of HCI and SE. Guidelines and code reuse are inadequate forms of reuse for this purpose. Design reuse in software has become a major issue, due largely to the concept of software design patterns. The pattern concept was developed in architecture in the 1960s and has been transferred to software design over the past decade or so. The level of interest in this paradigm from within industry and academia suggested that it would make a reasonable starting point towards design reuse in HCI. Patterns have turned out to be very suitable for HCI design reuse, and are central to the three aims of this thesis, i.e. reuse of high-level design, of low-level design, and combination of these approaches.

This chapter introduces patterns and the pattern language concept and their relationship to HCI. Section 3.2 describes the origin of patterns, their application to software design, how patterns are used, and the benefits and weaknesses of the approach. This overview of pattern languages would be of interest to both the software and HCI communities. In Section 3.3, I argue that, while patterns are useful for dealing with tension among traditional software attributes, they are even more suitable for HCI. The discussion in Section 3.4 summarises the chapter, emphasising that one aim of the present work is to produce pattern languages for HCI.

3.2 Theoretical Foundations of Pattern Languages

3.2.1 Historical Background and Essential Concepts

The design patterns approach was originally developed for town planning and building architecture. The primary driver was Christopher Alexander, an architect who also had a background in mathematics. Disaffected with modern architecture, he argued that a rigid design process in modern architecture had led to the

prevalence of buildings which were uncomfortable to live in and incapable of evolving. Alexander drew inspiration from ancient cultures, which had evolved buildings and town plans over generations [1].

Alexander observed that designs in a culture tend to converge over time. For example, he states the following about huts in Cameroon: "even superficial examination shows that they are all versions of the same single form type, and convey a powerful sense of their own adequacy and nonarbitrariness" (p.30) [1]. This type of convergence is a consequence of elements in the environment forcing designs in certain directions over time. When one observes the designs, one notices recurring features, which Alexander termed *patterns*.

If patterns were adopted as a central element of the design process, Alexander and his colleagues argued, then architectural quality would improve. He and his colleagues thus embarked on an effort to capture a set of patterns which would be of benefit to society. The result was a collection of 253 related patterns published in 1977 as *A Pattern Language* [4].

A pattern has certain fields. Every pattern must have a *Name* — a unique identifier that helps people to reference it and remember it. A pattern takes place in a certain design *Context*, and considers a recurring design *Problem* which occurs in this context. The pattern then analyses the problem, exposing the *Forces* which confront the designer. The pattern then describes a *Solution* — a proposed approach to the situation which resolves the tensions between forces. Consider Alexander's `A Place to Wait` pattern[4] . The context is any situation where people are waiting for something, such as a doctor's surgery. Two forces conflict: (a) patients must be present when the doctor can see them, but (b) the timing of this event is uncertain, leading to an anxious situation. A suggested solution is to draw in people who are *not* waiting. One hospital exemplified this pattern by building a neighbourhood playground which doubled as a children's waiting area, so that the young patients felt at ease before their consultation. `A Place to Wait` helps a designer appreciate what makes waiting places a problem, and gives practical advice for resolving the problem. It is *generative*, i.e. it facilitates the generation of design.

Patterns do not live in isolation. Once the solution to a pattern has been applied, a new context arises in which more detailed problems must be solved. Further patterns can be employed to capture the problem-solving processes inherent in this new context. A *pattern language* is formed when a collection of patterns is arranged in a hierarchical structure, with higher-level patterns yielding contexts which are resolved by more detailed patterns. This means that a designer can apply the pattern language generatively, beginning with a certain context, and working through all relevant patterns to generate the design. In `A Place to Wait`, the essential solution is to combine people who are waiting with others who are not waiting, and also to provide a quiet place where people can retreat while waiting. Alexander suggests several ways to achieve the first goal, by pointing to other patterns in the language, e.g. `Street Cafe`. The recursive nature explains why Alexander's patterns vary so widely in their granularity. The language begins with geographical distribution of society (`City Country Fingers`), works into town-planning (`Ring Roads`) and building architecture (`Staircase as a Stage`), and finishes at the level of detailed construction (`Paving with`

`Cracks Between the Stones`).

Patterns are not just a concept taken from architecture; they appear to serve a common need in many disciplines, even if known by names other than patterns (or by no name at all). Equivalents to patterns in civil engineering, such as "thin-core construction", have already been described in the previous chapter (Section 2.4). Brooks has also observed that medical practitioners have vocabularies to describe typical diagnoses and cures [32]. Coplien [55] has noted that Arthur Hall's work on system engineering acknowledged the critical importance of patterns before Alexander's work [1] even begun. More recently, Senge and colleagues have identified "archetypes" of organisations. These are typical situations which organisations face [202], such as the `Limits to Growth` archetype which relates to organisations which have "run into a brick wall" after a period of growth. It describes the forces which cause this situation to occur and outlines strategies for dealing with it. Another author, Kogut [127], has commented on the great amount of reuse made possible by the chemical engineering text, "Perry's Handbook" [95]. Among its benefits are the establishment of a common ontology, defining, for instance, standard types of heat exchangers (`air cooled`, `evaporators`).

Alexander's patterns can, in some respects, be viewed as an extension of the natural tendency for engineering sciences to define common terms for problems and solutions which recur. However, Alexander's work distinguishes itself by making the concept very explicit. Pattern languages are an overt attempt to better the discipline by defining common ground across different situations. Instead of acting as a background reference or a tool to check the validity of designs, pattern languages exist to promote a generative design process.

3.2.2 Pattern Languages in Software

The arena where pattern languages have made the greatest practical impact is software engineering. The movement began with Cunningham and Beck's pattern language for user-interface management in Smalltalk [17]. Jim Coplien was simultaneously studying "idioms" of C++ coding [52], programming language-specific patterns which represent recurring intra-class and intra-method structures. Coplien himself has noted that various people discovered the notion of patterns following their own experiences, and not by following the lead of Alexander or anyone else [56]. This suggests patterns are a natural phenomenon in software. It also may explain why patterns became so popular in software, once Alexander's work became a common ground for those who had noticed the phenomenon.

So what does a software design pattern look like? A straightforward example is the `Iterator` pattern contained in the well-known pattern catalogue by Gamma et al. [88]. A summary is shown in Figure 3.1 [1]. A pattern like `Iterator` works in a similar way to Alexander's patterns. It states the situation where it

[1] The form of this pattern has been adapted from that in the original so as to provide consistency with the rest of this thesis. There is more emphasis on the tension among forces which gives rise to the pattern and less on the elaboration of the solution.

applies and the problem to be addressed, explains the tensions which contribute to the challenge of the problem, and arrives at a solution which minimises the tensions.

Pattern 3-1. Iterator

Context: You are designing a system architecture. There is an aggregate class, i.e. one which represents a collection of objects (e.g. a tree, a list).

Problem: How do you enable the user to iterate through each object of the aggregate?

Forces:
- A common activity with aggregate classes is stepping through each item in the list, an activity which is required by algorithms for searching, sorting, processing, and so on.
- There must be some flexibility in the iteration order. With a list, for instance a user may want to step through in reverse. A tree can be traversed depth-first, breadth-first, etc.
- Iteration mechanisms can be complex, and there can be many mechanisms for a single aggregate class. The mechanism is further complicated by the need to keep state, i.e. the current position of the iteration.
- It should be easy to create a new iteration mechanism for an existing class. It should be easy to alter an existing mechanism.
- The aggregate class is likely to be complicated enough maintaining the constituent objects — it should not be burdened with extra complexity needed to support the iteration.

Solution: The solution is shown below. The chief innovation is the existence of a separate Iterator class. There is an abstract aggregate class and an abstract iterator class. This way, the client code can select the appropriate iterator for the task, and the aggregate is free of complicated iteration code.

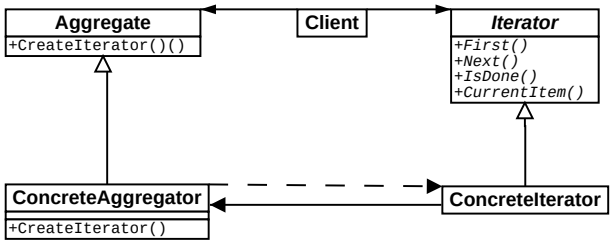


Figure 3.1. Summary of Gamma et al.'s Iterator Pattern (adapted from Gamma et al., 1995 [88]).

A problem in software design has been the tendency to focus on conventions and notation, rather than function. Students of design tend to learn how to draw class diagrams, and maybe how to identify classes from analysis. But few courses or textbooks cover what is meant by good design. The closest most come to this achievement is an explanation of design principles such abstraction, encapsulation, modularity, high cohesion, and low coupling. These are important, but offer limited value on their own. This situation corresponds closely to that of HCI guidelines, where principles such as *Feedback and Dialog* are difficult to dispute, but offer little practical value (see Section 2.4).

The power of principles comes with examples. Design patterns organise related examples in a mean-

ingful way, and show how to resolve tension created by trying to adhere to different principles can be resolved. In the case of `Iterator`, there is clearly a motivation to promote all of these principles. But instead of blindly stating them, the authors have shown how they apply to a common situation.

A striking contrast between building architecture and software architecture may explain why patterns have made an important contribution to the SE discipline. Building architects devote much of their training to the study of famous architectures and architects. Their software counterparts, on the other hand, are hard-pressed to name any famous system architectures or system architects [125]. Patterns are an excellent way to present successful designs to practitioners. Since a pattern is a summary of features which have been successful across different designs, it is able to concisely express ways to resolve tension among forces. This gives patterns a practical perspective which a traditional set of case studies could not achieve.

A further argument for the importance of patterns in software relates to software component reuse. This is a more established form of reuse, in which an existing code base is used in a new program. The ability to reuse code derives from the purely intellectual nature of software, which enables it to be acquired by one developer without another developer losing it. Reuse of software can enable a powerful form of incremental development, but can be hampered by poor documentation. Just as software users want to know how to use the software rather than how it works, so too do developers want to know how they can use the framework rather than how it works [122]. Patterns are a suitable way of showing how a developer can apply a reusable code base. Moreover, software components themselves are likely to be written with patterns of usage in mind. For this reason, Ralph Johnston has commented that frameworks represent both software components and design patterns [123].

Evidence of success with software patterns has been documented with Beck et al.'s coverage of several case studies [16]. These studies show that patterns have been useful in different ways, e.g. during design, in training, in documented knowledge encapsulated in legacy system design. A separate case study by Brown reaffirms the educational value of patterns [33]. These were case studies conducted by leaders of the field; while the patterns were used by pattern novices, the studies still do not necessarily reflect results which might be achieved in typical organisations. There are also some studies by practitioners, which speak largely in favour of the approach [128, 41], and considerable anecdotal evidence that patterns have become important to at least some parts of industry. Such evidence includes the popularity of pattern texts, especially Gamma et al.'s *Design Patterns* [88]. There are several annual conferences as part of the Pattern Languages of Programming series, with patterns also appearing in regular design journals and conferences. A large number of pattern meeting groups and mailing lists have emerged in recent years [106]. To summarise, there is evidence that patterns are having an impact in the software design world, although more investigation will be required to understand the full effects of patterns in the software industry.

3.2.3 Characteristics of Patterns and Pattern Languages

3.2.3.1 Importance of "Language"

The pattern concept has been discussed above without too much emphasis on pattern languages, for the sake of simplicity alone. It is the contention of this thesis that the "language" aspect of a pattern language provides much of the power of the patterns approach. Certainly, this is Alexander's view [2] and a perspective shared by many in the software design community. Gabriel outlined the significance of languages in *Patterns of Software* [86], and together with Goldman, is developing a set of patterns for Java's JINI technology [87] which work together in a coherent manner akin to Alexander et al.'s language.

To understand the importance of a pattern language, it is necessary to appreciate the subjectiveness of pattern languages. Far from being the objective and exhaustive catalogue of ideas they may seem to novices, patterns are based heavily on an underlying set of principles. Patterns demonstrate how forces are identified and resolved according to certain principles; in doing so, they are encapsulating a philosophy. Kerth and Cunningham have pointed out that Alexander identified, valued, and discarded patterns in a process which embodied his architectural philosophy [125]. A language is based on an underlying philosophy or set of goals, and it should generate artifacts aligned to the philosophy or goals. In this way, the artifact is derived from the patterns rather than directly from the philosophy. Gabriel [86] has likened this phenomenon to that of a tennis player learning to hit a ball. The underlying goal is to hit smoothly and powerfully, but the instruction is to aim for a point just beyond the ball. There is no obvious connection between the goal and the instruction. Yet, following the instruction achieves the goal. Ideally, a developer should be able to resolve forces by thinking about patterns rather than relying on abstract principles.

Pattern languages are the missing ingredient required to achieve this goal. An individual pattern will be limited in scope, and few projects are small enough to derive large benefits from one pattern. Instead, most projects warrant a "team" of patterns, aware of each other and working together. To explain how a good pattern language can work in practice, consider the following scenario. Software developers begin a program design by applying the solution of pattern A. This leads to a lower-level design problem, which pattern A indicates is resolved by applying either pattern B or pattern C. The designers settle on C because it fits their needs. Applying pattern C to the existing design is straightforward because pattern C assumes pattern A has been applied where necessary. Several months after coding has been completed, a change request arrives. The designers realise pattern D would be helpful, and pattern D is again straightforward to implement because it is based on the same concepts as the previous patterns.

As an example of coherence, reconsider Alexander et al.'s `A Place to Wait`. The pattern helps to resolve situations left open by other patterns, such as `Health Center` and `Office Connections`. The pattern itself suggests further patterns: `Street Cafe`, `Opening to the Street`. Two important points are exemplified here. First, the language allows a generative design process. Each Pattern solves part of a problem, and this leads to new problems which can be delegated to further patterns. The second

point, equally important but often overlooked, is that the patterns are mutual in their underlying values. In Alexander's case, he apparently has certain principles which he would like his language to accomplish; a sense of community ownership, a mixture of work and play, and a capability to spend time alone, to name some of the principles which recur throughout his patterns. Yet, it is conceivable another author may have a different perspective. Imagine if a pattern language for town planning were devised by an author arguing for more time together among families. A language like this may claim each home should be a health center, with medical staff being mobile, or that street cafes should be abandoned in favour of home cooking. Where would `A Place to Wait` fit in? Clearly, it would be inappropriate because it supports a different set of values. Even if waiting places are needed, the pattern would need to be revised. Perhaps housing would be combined with waiting places so that waiting could take place in the home. The point is that a pattern language based on mixed values leads to a confused design which achieves no-one's goals. In contrast, a language where all patterns are based on the same underlying values leads to a coherent design which exemplifies the underlying values.

An important consequence is the Alexandrian notion of "piecemeal growth". That is, gradually changing the structure to adapt to the needs of its inhabitants. The software analogy is responding to change requests. The goal in both cases is the same: to make the transition harmonious. Alexander argues that piecemeal growth is difficult to achieve in modern architecture because of the rigid design structures commonly employed. In software, the problem is all too familiar — changes can be very difficult to implement once a system has already been coded, due to the large number of interdependencies often present in a significant system. Cohesive pattern languages would assist software developers in making smooth incremental enhancements. The example above, in which pattern D was able to be adopted several months after applying patterns A and C, illustrates the point. Patterns A, C, and D are aware of each other and follow the same principles. Thus, pattern D, required some time after users began interacting with the system, will merge well with the existing system because it was created using patterns A and C. This is what Alexander meant by "piecemeal growth".

These arguments demonstrate that patterns are most powerful when they form components of cohesive pattern languages. It should be noted that many existing approaches to pattern languages do not work this way. One approach is to publish a single pattern — many papers of this nature have appeared at the major patterns conferences. The second approach, certainly an improvement, is to publish a set of patterns related in scope. For example, a set of patterns which may be used for networking or for graphics. In Gamma et al. [88], probably the most widely-known patterns reference, the authors acknowledge they have put together a *catalogue* of patterns, rather than a *language*. They acknowledge that Alexander suggests an order for applying the patterns and comment that his work is different insofar as it claims to create a completed product. Certainly, in their case, there are areas where the patterns do work together in a coherent manner. For example, the authors explain how the model-view-controller architecture represents a number of patterns in their collection. Gamma et al. explain that some collections can be more like

languages than others, and they have produced something in the middle of the spectrum. The success of their work indicates that a highly-cohesive language is not a prerequisite to a useful language. However, the work in this thesis does pursue the concept of a well-integrated language because: (a) even though a collection can be useful without being a language, the greatest benefits of patterns nevertheless exist when they are used in a language, and (b) pattern language theory seems to match closely to HCI concerns (see Section 3.3), suggesting that HCI pattern languages may be more feasible than software pattern languages.

3.2.3.2 How Patterns are Captured

Patterns are discovered, not invented. A good design contains patterns worth capturing and documenting. Richard Gabriel summarises the situation [57]:

> *If you walk into a room and you ask Alexander to list all the patterns he sees, he will not look for sheets of paper with patterns written on them, he will look at the room and tell you the ones he sees in the spatial configuration. Similarly, if asked what patterns there are in a software system, an astute patterns person will look at the code and try to list them off.*

A pattern language should be based on observations from successful, real-world systems. There are various ways to discover patterns —- three valid approaches are described by Kerth and Cunningham [125]:

Introspective Patterns reflect the experience of the author.

Artifactual Patterns reflect systems surveyed and analysed by the author.

Sociological Patterns reflect author's analysis of developers as they solve problems.

In order to be a cohesive language, it is important that the observed systems have a similar scope and similar underlying principles. Then the design features among the systems will have a greater affinity with each other, a property which will be evident in the relationship between resulting patterns.

3.2.3.3 Presentation of Patterns and Pattern Languages

Patterns are typically represented in what is called *Alexandrian* form, after Alexander et al.'s original format [53]. This format consists of several fields. The major fields of a pattern — Name, Context, Problem, Forces, Solution — have already been identified. In Mezaros and Doble's guide for writing software pattern languages [164], the five fields above are classed as mandatory. Several other fields are considered optional and may be used for a particular pattern if the designer chooses to do so. These are: Indications (that the problem exists), Resulting Context, Related Patterns, Examples, Code Samples, Rationale, Aliases, Acknowledgements. In addition, patterns by Alexander and others sometimes include an Image and a validity ranking.

The author of a particular language can choose a set of fields which are appropriate for the language. They do not have to be identical to those above, as long as they are consistent throughout the language. Patterns authors in software have preferred this flexible approach, although it is possible more standardised formats may emerge for HCI patterns (see Section 4.3.3).

Some patterns are presented in a style like Gamma et al.'s [88], which explicitly states each field. Others follow Alexander et al.'s lead and distinguish between fields via formatting aids, such as boldfacing and spatial relationships. The latter style tends to be more literary and may engender a greater sense of flow between sections. It may be better for conveying the feeling of the language, and may therefore have a longer-term educational perspective. However, this thesis has adopted the Gamma et al. style because its explicit format makes the distinction among fields clearer.

While defining pattern structure, the structure of languages should also be considered. Patterns within a language can be related to each other in different ways. Zimmer [231] has placed relationships among Gamma et al.'s patterns [88] into three categories: X uses Y in its solution, Variant of X uses Y in its solution, X is similar to Y. The key message here is that some patterns delegate to other patterns, and some patterns also "compete" with other patterns, i.e. have a similar context and/or problem, but approach it in a different way. The designer must have a strong enough understanding to weigh up the options and determine the best approach.

3.2.3.4 Benefits

Several major benefits have been ascribed to pattern languages, and are described below.

3.2.3.4.1 Patterns Aid Design The generative nature of patterns can support design better than working from principles alone. Because they are based on proven design concepts, patterns help to ensure higher design quality at an earlier stage, with less need for iteration within the design process. Good patterns also contain useful examples drawn from real-life projects. This is of great benefit to designers, whose training often lacks such stimulus (unlike building architects, as discussed earlier). A well-written pattern language should enable a design to flow forward from a high-level specification to a detailed design artifact.

3.2.3.4.2 Patterns Aid Redesign As well as providing support for initial design, good pattern languages also support redesign. This is a consequence of the coherence of patterns languages, which is supposed to lend itself to "piecemeal growth", as described in Section 3.2.3.1. Vlissides has claimed that patterns support redesign even when the initial artifact did not derive from patterns-based design [226]. this fits well with Gabriel's statement (Section 3.2.3.2) that an astute patterns person can look at code and list off patterns, even in the absence of explicit labels. As long as designers use patterns that are compatible with those (possibly implicit) patterns existing design, refactoring provides at least some of the value of an overall patterns-based design process.

3.2.3.4.3 Patterns Capture Design Philosophies Alexander's major motivation was to capture the "timeless" knowledge encapsulated in traditional architectures, and this goal has been retained with many software patterns. Kerth and Cunningham have observed that patterns can ensure a consistent approach among developers in different organisations, such as Macintosh Developers [125]. It is worth speculating about what a pattern language for direct-manipulation GUIs would look like. It would presumably encapsulate information about drag-and-drop, buttons, selection mechanisms, etc. There would be rich interaction among these patterns. Here are messages which three patterns might convey: (1) After an object is "drag-and-dropped", the object remains selected; (2) A button acts on a selected object; (3) Clicking on an object selects it and removes the previous selection. The patterns would probably be somewhat broader in scope and outline the rationale for these decisions. Such a pattern language would then be a coherent statement of how the mechanisms relate to each other. It would tell a designer how to adhere to the mechanisms. Underlying each pattern is a consistent design philosophy, a set of principles which certain designers feel will optimise the user experience. The language would express what the design philosophy means in real-life design contexts. Supplementing a set of principles with a set of patterns forces designers to consider, and convey, how the principles apply to typical design problems which arise. In this sense, pattern languages aid not just immediate design; they also act as a means of long-term education.

3.2.3.4.4 Patterns Capture Expertise Across Diverse Contexts A feature of Alexander et al.'s pattern language is the diversity among the patterns. The language transcends international boundaries and historical eras, often venturing to cover an observation from several cultures in a single pattern. Furthermore, relevant literature is cited as additional support for the ideas embodied in each pattern. The pattern format is a suitable way to package scattered knowledge into a tool which is immediately applicable to designers. This is particularly important in software engineering and HCI, where the pace of innovation is so rapid that it is often difficult to transfer knowledge from one technology to the next.

3.2.3.4.5 Patterns Provide a Common Vocabulary Each pattern in a language has a distinct name. The name provides a common label for a construct which designers may have seen before, but could not express easily. Alternatively, encountering the label may be a cue for a designer to learn more about the notion it refers to. This property implies that patterns "effectively expand people's communication bandwidth" [226], an important achievement when so much communication occurs during design projects.

When people are mutually aware of certain patterns, they can use statements like "I have used an `Adaptor` here". This is preferable to "I have created an intermediary class with the interface expected by the calling class, which delegates the real work to a third implementing class". Beck has observed the phenomenon of patterns gradually becoming part of a vocabulary for a team recently introduced to patterns [16].

3.2.3.4.6 Patterns Provide Design Rationale and Support Design Documentation Because patterns are based on successful design ideas, an effective incorporation of patterns into a design can help to justify the design's efficacy. The pattern's examples, in particular, will allow designers and reviewers to reason by analogy; by stating the patterns used for a new design, previous systems using the same pattern can also be identified, i.e. the pattern's stated examples. Designers can then ask how the lessons of those previous designs apply their new design.

The common vocabulary offered by patterns also improves documentation clarity; people already familiar with the pattern will have insights into whether it is appropriately used in the new context [19]. Of course, patterns may be present in a design even though they have never been explicitly documented; this follows Gabriel's [57] comment (Section 3.2.3.2) that each software artifact contains a number of patterns, implicit or otherwise. The explicit identification of new patterns in a design will still be useful, since it improves the clarity of the design by alerting the reader to a pattern which has been used effectively in previous systems (but never documented as such). Inclusion of such patterns is also valuable as it should facilitate a well-considered argumentation of the design.

3.2.3.5 Limitations

There are several limitations which must be balanced with the benefits of patterns.

3.2.3.5.1 Patterns Do Not Automatically Generate Quality Designs One misconception of the patterns concept is that it allows people to designs effectively simply by following a rigid, rule-based, approach to design. In fact, patterns act as a resource for designers; the designer must still choose appropriate patterns, tailor patterns, apply patterns, and decide when patterns are inappropriate altogether. In Alexander's words (p.206-207) [2]:

> *The rules of English language make you creative because they save you from having to bother with meaningless combinations of words ... A pattern language does the same.*

Patterns are no silver bullet for design; they simply represent a tool for designers to apply existing knowledge to new situations.

3.2.3.5.2 Patterns Are Imprecise For patterns to be of practical benefit, information must necessarily be simplified and generalised. This can present a problem to researchers who demand precision [76]. The quality of patterns can be difficult to assess due to their imprecise nature. This suggests empirical studies may be the best way to verify the usefulness of a particular pattern language. The present thesis contributes in this respect, by performing studies with designers who apply techniques for reuse.

3.2.3.5.3 Patterns Require an Appropriate Process Patterns represent a change in thinking with respect to the design process and communication among designers. As a high-level representation of design,

patterns in design may also be of interest to other development activities, e.g. software testers could use the patterns to inform the generation of test cases. All of this suggests the development process may need to be rethought for patterns to be truly effective. Work in this area has been limited to date.

3.3 The Argument for Patterns in HCI

There are empirical and logical arguments to demonstrate the efficacy of patterns in software design. Patterns can help designers grasp the meaning of principles like modularity and encapsulation, and readily apply them to their everyday work. Designing according to these principles improves maintainability, reliability, performance, and other traditional software attributes. But this thesis is concerned with user-centered design, and Chapter 2 has already explained that traditional software attributes are often in conflict with usability concerns. It is not surprising, then, that authors of detailed software design patterns have generally not paid much attention to usability.

When Alexander claimed patterns would lead to *habitable* environments, he referred to the people living in an environment. The software patterns community has generally viewed habitability as belonging to the developers, so that the goal of design patterns is to make developers feel comfortable as they maintain and upgrade their work material. With usability, the focus shifts towards the experience of end-users. This is a major shift, and it can therefore not be taken for granted that the success of detailed software design patterns will translate to patterns of HCI.

In fact, some analysis suggests that patterns do align closely to HCI concerns, and several factors suggest they may offer even more to HCI design than to detailed software design. This is true of Alexander's original ideas for pattern languages, and also applies to many of the arguments for patterns in traditional software design.

Alexander's patterns strive for what he labels the "Quality Without A Name" (QWAN). This quality is ethereal according to Alexander, but he nonetheless likens it to several other terms: alive, whole, comfortable, free, exact, egoless, eternal. These terms represent much that is desired in interactive systems, and resemble terms associated with usability, such as harmonious, transparent, satisfying. It is also important to realise what these terms are *not*. In architectural patterns as well as usability, there is a general agreement that aesthetic beauty is only a small part of the picture. Developers of IBM's Common User Access guidelines concluded that "only around 10 percent of a product's usability comes from what you see on the screen" (p.52) [195]. The remaining 90% is composed of interaction mechanisms and the underlying components . Likewise, architectural patterns ascribe great value to the user experience: "a building is as much the life that goes on inside as it is the "shell" which encloses the life" (p.55) [92].

Coplien has stated that the humanistic side of patterns defies many software engineers [55], yet this aspect of patterns is perfectly in tune with an HCI perspective. In a keynote presentation at OOPSLA '96 [3], Alexander questioned whether features of his patterns had been translated into software. He explained

that patterns may make software “better”, but that did not mean the moral component had been transferred, prompting the question: “Will it actually make human life better as a result of its injection into a software system”. He went on to explain “I don’t pretend that all the patterns that my colleagues and I wrote down in *A Pattern Language* are like that ... But, at least it was the constant attempt behind our work”. While software patterns may impact on the work quality of software developers, HCI patterns have much broader scope for impact. In this sense, they fit better with Alexander’s original view of pattern languages.

High-quality usability adheres to the well-known principles of usability, such as minimising cognitive load, placing the user in control, and so on. The generative nature of pattern languages helps to deliver these principles by showing what they mean to real design situations, just as detailed software design patterns do the same for software designers. Figure 3.2 provides a simple representation of the situation. The space of concrete design solutions is supported by a number of principles which are present across the entire space. There are examples of good design at certain points in the solution space, representing solutions which resolve tension among the principles at that point. A pattern is drawn from several successful instances. The pattern should be able to generate new solutions which fit inside the cluster. The diagram shows that patterns represent an intermediate form of reuse, between global principles which may be too vague to use and individual examples which may be too abundant to learn in detail, and may not readily transfer to new situations.

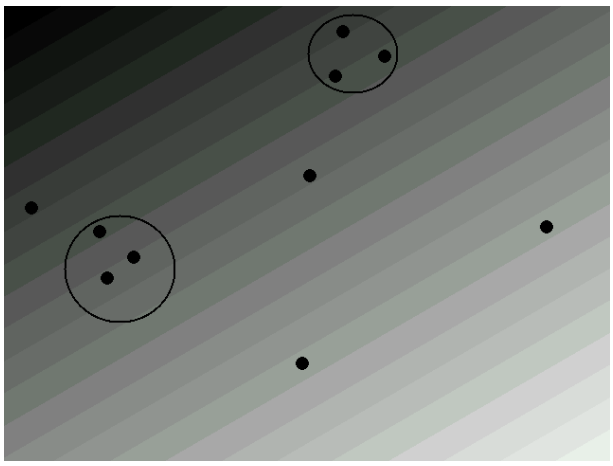


Figure 3.2. A perspective on the relationship between principles, patterns, and examples. The entire space is covered by a set of desirable principles. Examples are shown as black dots, each representing a single instance of a design feature which resolves the tension among the prevailing principles. Patterns are shown as clusters of related examples.

The goal of piecemeal growth, inherent in patterns research, resonates strongly with HCI concerns. In HCI, as discussed in Chapter 2, there is a great need for iteration. This is true to a degree in software generally, but particularly pertinent when usability is a high priority. It is true on two levels. Firstly, the

lack of psychological knowledge means users' reactions cannot be effectively predicted *a priori*; there is a need to feed knowledge about user experiences back into the target artifact. Secondly, there is the task-artifact cycle [37]: a tool may be designed with a certain set of tasks in mind, but users generally create new tasks which had not previously been considered. This in turn necessitates redesign and a new tool is built. The cycle continues. The process is self-perpetuating, and implies that even if we could accurately comprehend user psychology, we still need to design iteratively.

Fundamental to patterns is the process of participatory design. This is a concept which, like patterns themselves, began in architecture and has since been applied to an HCI context (see [170]). In architecture, the goal was for the community as a whole, in particular the eventual inhabitants, to work together with practitioners according to the language. In HCI participatory design, various stakeholders, in particular the end-users, work with development staff to design the system. This approach to HCI is important due to the interdisciplinary nature of interactive system design, which can draw skills from psychology, computer science, graphics, marketing, linguistics, and so on. Patterns can support interdisciplinary design primarily by forming a common language across disciplines.

Erickson has illustrated why the "Lingua Franca" property of pattern languages may be a very powerful one [74]. He describes a town-planning project to improve tourism for a small town in which important locations were listed by the residents. The list became known as the "Sacred Structure" and the places later became known as "Sacred Spots". This shared language enabled the townspeople and developers to hold common beliefs and values, leading people to work towards common solutions. Erickson's colleagues have also explained their vision of work in the future, whereby patterns would be a central way of defining and directing a company's directions, so that workers make decisions consistent with the overall objectives [102]. If HCI design features had common names and definitions, then people from a variety of disciplines could communicate more smoothly. A software engineer would more easily comprehend the task which must be achieved and perhaps estimate its complexity, based on previous attempts to implement such a pattern. An end-user could recognise links between a proposed feature and features of previously-encountered software. And so on for other stakeholders.

It was mentioned above that patterns enable expertise to be drawn from a wide range of sources. These sources can span not only current software, but also software from earlier times, and even artifacts which are not digital at all [219]. Again, this property seems more pertinent in HCI than detailed software design. In HCI, the human is considered as well as the computer. Thus, compared to software design patterns, there is a key element of the design problem which is consistent across many diverse contexts. Thus, HCI patterns can draw from experiences with digital watches, microwaves, and many other devices in addition to conventional desktop computers.

The popularity of detailed software design patterns also suggests a further, pragmatic, motivation for HCI patterns. While HCI patterns and software design patterns are not the same thing, they do actually overlap significantly; HCI patterns effectively lead to a "conceptual" or "high-level" design, which in turn

can be supported via software design patterns. By devising techniques to capitalise on this overlap, for instance by relating the HCI patterns to useful software patterns, a synergistic effect is possible. Thus, HCI patterns are more attractive due to the existing application of design patterns. A combination of this nature is a key goal of this thesis.

3.4 Discussion

A pattern is a form of expression which captures a successful design solution to a significant problem. A pattern language extends the notion by documenting a series of related patterns. Higher-level patterns solve part of the problem and delegate the remainder to lower-level patterns. Ideally, the components of a language share a consistent set of underlying values. This way, design can progress smoothly, and ongoing change requests can be supported.

Pattern languages introduce many concepts which make the approach appropriate in an HCI context. A key contention of this thesis is the crucial nature of the *language* in *pattern language*. In their most appropriate use, design patterns combine to form a language. The language is *iterative*: High-level patterns delegate detailed decisions to low-level counterparts. The language is *coherent*: All patterns form a team which pushes the design in a certain direction, so that an underlying set of values is achieved by following the patterns. The idea of a language is associated with a number of strengths of the pattern language concept. Following is a summary of the key concepts and terms introduced by Alexander.

Quality Without A Name The underlying values of the pattern language are intended, at least in Alexander's work, to lead to a particular quality which may be likened to terms such as alive, whole, comfortable, free, exact, egoless, eternal.

Generativity Patterns help designers move forward in their designs. This is a more effective than making haphazard attempts and evaluating the result according to design principles. Generativity is provided by the structure of the patterns and the language. The patterns indicate where they apply, outline the tensions involved, provide a solution, and then identify other patterns in the language which might be used to proceed.

Piecemeal Growth A goal in pattern languages is to achieve smooth, flexible, growth, which follows from the conviction that quality artifacts result from continual modification rather than a single upfront design effort. It is the coherence of the language which facilitates piecemeal growth. A system may be designed with several patterns. A few years later, more patterns might be applied. Because the patterns are aware of each other, and have the same underlying values, redesign should proceed in a relatively straightforward manner.

Habitability Pattern languages in architecture seek to achieve something more important than aesthetic beauty; the aim is improve the quality of those who live in the target environment. In software

patterns, this is generally thought of as the developers who maintain code, and in HCI, the experience of the end-users is primary. Artifacts are habitable when they exhibit the Quality Without a Name and can be adapted organically according to people's needs, i.e. follow piecemeal growth.

Timelessness A strength of pattern languages is they abstract from the details and capture knowledge which is applicable across diverse times and contexts.

While these aspects of pattern languages have not always been translated fully into software design, the basic concept of patterns has nevertheless proven very useful in that domain. Among the benefits for pattern languages are: aiding design and redesign, capturing design philosophies, capturing expertise across diverse contexts, providing a common vocabulary, providing design rationale, and supporting documentation. These benefits follow logically from the nature of pattern languages. There is a large amount of anecdotal evidence which suggests patterns are being applied in industry, and successfully so. However, more rigorous research will be required to understand whether patterns are having a meaningful impact in organisations.

The patterns concept promises to deliver benefits in various disciplines, and the case for patterns in HCI seems particularly strong. The original ideas about pattern languages were about the life quality of inhabitants, similar to the focus in HCI on end-users. This is the driving force behind the ideas embodied in the Quality Without A Name. The notion of piecemeal growth resembles the iterative lifecycles which have been a recurring theme in HCI literature, as was discussed in Section 3.3.

A critical point here is that many of the benefits are especially strong in the context of pattern languages; a collection of isolated patterns would have limited capacity to produce these benefits. Therefore, it is clear that if HCI patterns are to be most effective, it is languages of related patterns that must be produced.

The previous chapter argued in favour of reuse, but did not cover the mechanism for achieving it. This chapter has explained that the pattern language idea presents itself as a suitable mechanism for reuse in HCI, but "pattern languages for HCI" is still a very abstract concept — I have not yet explained what patterns could be possible or how they might look. There have been several recent approaches to patterns in HCI, including my own work. The next chapter describes recent approaches and explains how the present thesis compares with other work.

Chapter 4

Applying Patterns to Human-Computer Interaction

4.1 Introduction

The previous chapters have argued that patterns, on face value, promise to deliver improved design reuse in HCI. However, actual attempts to use patterns have not yet been discussed. This chapter outlines existing research approaches and provides a context for my own research, which is covered in detail from the next chapter onwards.

This chapter begins by describing several dimensions of pattern collections, in Section 4.2. The analysis is restricted to HCI-related patterns, even though it could be argued the classifications apply generally to pattern collections. The classification helps to understand the contribution of pattern-related HCI research projects. Furthermore, it provides the basis for an explanation of the work conducted in this thesis, and its place within the landscape of HCI-related patterns. Section 4.3 summarises twelve significant HCI pattern collections, and classifies each according to the dimensions of Section 4.2. Some important contributions from researchers have been *about* HCI patterns rather than directly contributing a collection of patterns. Section 4.4 summarises recent research of that nature.

The present thesis is introduced in Section 4.5. In this section, I note that flexibility in some dimensions will lead to rigidity in other dimensions. To produce an interdisciplinary HCI pattern language, compromises must be made along these dimensions. Section 4.5 introduces the pattern languages developed during this thesis, and explains how they have been developed with deliberately tight scope in some respects to allow for broader scope in other respects. Section 4.6 summarises the chapter and points to the remainder of the thesis.

4.2 Dimensions of Pattern Collections

A 1997 workshop on HCI patterns suggested that patterns for interactive usability could have various goals and manifest themselves in different forms [14], and this view has been vindicated in the ensuing years. This section identifies three dimensions of pattern collections: level of abstraction, target medium, specialised requirements. The dimensions are based on observations of HCI pattern collections from other authors as well as my own work. It is not possible to claim the dimensions are orthogonal; the classification can be no more exact than the pattern collections themselves, and none of HCI, pattern collections, or HCI pattern collections, are particularly well-defined. Each dimension is nevertheless intended to offer an additional degree of freedom. Later on, it will be shown that the various HCI pattern languages are diverse with respect to the dimensions.

While all dimensions are important, most attention is paid to the first level of abstraction, because this dimension was considered in most detail early in this project. A paper based on this work was presented, primarily to suggest possibilities of patterns [145], and has since been used in at least one pattern language to distinguish between the various patterns [186].

4.2.1 Level of Abstraction of Patterns

HCI is a broad area, involving a variety of constructs: tasks, user-interface objects, and so on. Patterns can potentially represent each of these. The situation is analogous to software design patterns. Buschmann et al.'s text, for example, contains three types of patterns [34]: (a) architectural patterns (e.g. Client-Server), which cover the overall architecture of a system, (b) design patterns, which typically explain how several classes relate to each other (e.g. Publisher-Subscriber), and (c) idioms, which are coding patterns specific to a programming language (e.g. counted pointer). In this section, five levels of abstraction are suggested, with my own example for each level. The five levels are not intended to be exhaustive; other levels of abstraction, such as patterns of process and patterns of functionality, are also possible. Some of the levels of abstraction were originally taken from a paper by Casaday [39], in which several candidate pattern types were proposed.

4.2.1.1 Patterns of Tasks

Tasks play a vital role in HCI. An understanding of tasks that users will perform gives developers an insight into the functionality which should be provided and how it will be used [187]. Because most stakeholders can understand task descriptions to some degree, they also constitute a powerful communication tool, with artifacts from task analysis acting as a common ground for developers, client representatives, and users alike (see [12]). If this advantage is to hold true, tasks should be labelled consistently. The patterns concept suggests itself as a logical way of looking at tasks, as well as imparting information about how they may be supported.

`Open Existing Artifact` (Figure 4.1) demonstrates the range of levels a task pattern language might address. Task analysts can use patterns of typical tasks to identify and discuss tasks which a system should support. Techniques for providing this support can also be captured by a pattern. In `Open Existing Artifact`, the advice is quite high-level; it suggests simply to provide cues about file contents. However, if this kind of language were customised to the needs of a particular project, then it would be possible to cover issues as detailed as human-machine dialogue, screen appearance, and software design. If we knew that the pattern applied to the selection of an image, we might suggest a text selection list alongside a frame of thumbnails. The environmental context in which the action takes place could also be considered, as well as the user's cognitive processing as the task is performed.

There is a rich inter-relationship among such task patterns. The rules of a task pattern language could guide how tasks fit together. A `View Artifact` pattern, for example, might propose that the user be able to open the document, browse its contents, make searches, and so on. Thus, it resolves some of the forces itself and delegates the remaining issues to other patterns, just as Alexander's language does. A sentence in this language is a sequence of tasks which the user performs. A task analyst can use the language's grammar to shape the user's experience with the system. Because task patterns are small units, they apply to, and may be drawn from, many contexts. In this case, the `Open Existing Artifact` task is supported by word-processors and web search engines alike. Obviously, there are differences in the way the task is performed, but there is nevertheless much to be gained by surveying how various systems support a given task and abstracting this information.

Pattern 4-1. Open Existing Artifact

Context: Any software which allows the user to view or edit different documents should provide a mechanism for the user to request a previously- prepared document to be opened. The artifact might be a text file, an image, a database, even a state-vector for a simulation or game. In some cases, such as when a web search is performed, the user may not be searching for a specific document.

Forces:
- The user is required to identify a particular artifact.
- There may be many artifacts to choose from, especially in a networked environment.

Solution: A document's name or location alone is often not a sufficient way to identify it. The system should present cues about the document's contents to support the decision-making process, since the contents are likely to be more easily-recalled than an arbitrary name or internet address. Examples include showing the first few lines of text documents or, in the case of image files, providing thumbnails. Allow the user to search for contents. For efficiency, provide shortcuts to files which user is most likely to be interested in, e.g. keep a list of recently-opened artifacts.

Figure 4.1. Example sketch of a task pattern, Open Existing Artifact

4.2.1.2 Patterns of Users

Developers need to acknowledge individual differences of users such as frequency of use, general experience with computers, and domain expertise [211]. As `Intermediate User` (Figure 4.2) shows, a user profile pattern can be used to explore the forces involved in the context of a particular kind of user accessing the system, and to specify the user-interface accordingly. In this case, the pattern concerns user expertise. Other characteristics, such as user role (e.g. system maintainer, entry clerk), could also be addressed. Because this form is less specific about functionality than task patterns, it is unlikely to aid detailed software design.

Marketers frequently divide the market into manageable segments, based on demographics and other variables [40]. A user profile pattern can be seen as a way of solving the marketer's problem of catering to a particular market segment. In this way, such patterns might contribute to improved coordination between marketers and usability specialists.

Pattern 4-2. Intermediate User

Context: Any system where a user is likely to use the system long enough to make the transition from novice to intermediate.

Forces:
- Software should provide substantial assistance to novices.
- Educational aids can distract, obstruct, or irritate users as they become more fluent. At the other end of the the spectrum, advanced features may be difficult for intermediate users to discover and learn about.

Solution: The intermediate user is comfortable with working in the regular environment. Dialogues with "Don't ask me again" checkboxes and other mechanisms should be used to help the user customise their environment without having to enter specialised modes (see [47]). Where possible, the system should allow them to stay in their regular working environment when they are customising the system [46].

Figure 4.2. Example sketch of a user profile pattern, Intermediate User

4.2.1.3 Patterns of Inter-User Interaction

A rather different approach might be an analysis of the way people interact when performing Computer-Supported Collaborative Work (CSCW). One possible formulation is exemplified in the `Discuss an Artifact` pattern (Figure 4.3; based on design decisions in the CAR system [201]). Resembling the task patterns discussed previously, this type of pattern addresses styles of interaction. Again, there is the potential to create a language rich with structure and information, which would help users to communicate as freely and effectively as possible. Such a language would need to fit in with organisational principles, and may resemble the envisioned usage of patterns as suggested by [102] (discussed in Section 3.3).

Pattern 4-3. Discuss an Artifact

Context: A system where physically-separated users share the same screen contents and need to discuss an on-screen artifact.

Forces:
- Users sometimes need to use a local area, invisible to other users.
- Users will be concentrating on the communication and may experience confusion about the distinction between their local area and the shared area.
- A confused user will not only experience personal frustration, but will also slow the group's progress.

Solution: Provide a *local* pointer icon for private usage, and a *global* pointer icon to alert the user that everyone can see it.

Figure 4.3. Example sketch of a multiple-user interaction pattern, Discuss an Artifact

4.2.1.4 Patterns of user-interface elements

A seemingly obvious type of HCI pattern collection is a set of user-interface elements, or "widgets". The purpose would be to help detailed designers and programmers understand where it is appropriate to use a certain user-interface element, possibly as a replacement for traditional documentation on toolkit usage. However, as `Scrollbar` (Figure 4.4) shows, this type of pattern may be rather contrived:

- The scrollbar has been selected as the solution, and the problem reverse-engineered. The process fails because one problem can have many solutions and vice-versa. Another solution to the problem, for example, would be to scroll the document when the mouse pointer is positioned at the boundary. Even when there is one solution, more than one widget is often required. A thumbnail view of the entire area could supplement scrollbars.
- Frames and other user-interface elements do not fit any context specified in user-centred terms, because they primarily exist for programming convenience and are invisible to the user.
- One pattern, with a solitary problem to solve, is not enough to specify all of the widget's features.

The above discussion suggests that patterns of individual widgets are ineffective. Technically, a widget may be seen as a design pattern, insofar as it is a common solution to a common problem. However, the utility of such a pattern is questionable. The solution to a problem is likely to be obvious whenever the solution is merely to use a scrollbar, a checklist, and so on.

There might be some benefits if the widget is somewhat novel. For example, if a company produced a new toolkit, patterns at this level could suggest where it is appropriate to use certain widgets.

In a general design context, though, developers are familiar with individual widgets, and the complexity of the user-interface design task is mostly due to problems in combining widgets effectively. A more

Pattern 4-4. Scrollbar

Context: User's working area exceeds the area which can be displayed at one time.

Forces:
- The user needs to switch between different parts of the artifact.
- The artifact is often too large for the display device to clearly represent it at one time.

Solution: Associate a scrollbar with each dimension of the viewable area which is smaller than the corresponding dimension in the working area.

Figure 4.4. Example sketch of a user-interface element pattern, Scrollbar

fruitful approach might therefore be a collection of inter-widget relationships, since well-designed software often arranges widgets in useful ways. The status object, described in `Show Status` (Figure 4.5) is an arrangement which is straightforward to implement, but can substantially boost learnability.

This pattern language leans further towards the detailed design end of the spectrum. The concrete nature of patterns means they are an appropriate way to capture relationships between widgets, and this represents an important opportunity to provide usability-oriented advice in a coder's own language. At the same time, it would be necessary to consider how higher-level issues can be addressed. Since the patterns concern the user-interface, there is a question as to whether they can be related to the constructs covered in previous sections, e.g. users and tasks.

Pattern 4-5. Show Status

Context: A system where objects' internal states are not entirely obvious by appearance alone.

Forces:
- The user should be able to infer information about internal state by observing external appearances.
- External appearance must not show too much, lest the interface be cluttered and overwhelm the user.
- There is often not enough room to show information about internal state.

Solution: Provide a status object whose appearance depends on the object under the mouse pointer. Gamma et al.'s `Observer` pattern can guide the implementation of this facility[88].

Figure 4.5. Example sketch of a user-interface element arrangement pattern, Show Status

4.2.1.5 Patterns of entire systems

Patterns of entire systems capture the issues involved in their development. Characterising a set of related programs into an individual pattern is a non-trivial matter. In Casaday's proposed language, each pattern

represents a system with a set of desirable usability attributes (e.g. memorability) coupled with external factors which might constrain the solution [39]. For example, Casaday's `Airport Passenger` is a pattern for systems to be used by customers at airports. Consequently, its key attributes are efficiency, reliability, and immediate learning. The solution in this case is to standardise components and procedures where possible. People can then arrive at a new airport and use the system quickly and with low chance of error.

Another way to classify systems is according to their purpose. Systems which manipulate documents, such as word-processors, might be represented by something like `Document Manipulator` (Figure 4.6). This kind of pattern would be a useful way of presenting analysts with the general issues involved in producing certain kinds of systems and if the pattern represents a very specific type of system, the solution could also include detailed design information.

However, describing an entire system in one pattern can pose severe limitations. As discussed in Chapter 3, a system should be viewed as a whole bunch of patterns interacting together, rather than as one big pattern. `Document Manipulator`, for instance, should be as relevant to a drawing tool as a word-processor. To satisfy this requirement, much useful detail must be abstracted away. The major elements of a word processing document come from the keyboard, rather than a tool palette, which is what `Document Manipulator` would imply. `Document Manipulator` could be split into more specific patterns (`Word Processor`, `Drawing Tool`), but substantial redundancy would result. The only way to resolve the redundancy would be for each of these pattern to delegate to more detailed patterns. Thus, even if patterns of entire systems arwe used, they must be combined with patterns at the sub-system level.

Pattern 4-6. Document Manipulator

Context: The user needs to arrange a set of related elements together to form a document.

Forces:
- The user should be able to efficiently build and edit documents.
- The internal representation does not match the user's perception of the document being worked on.

Solution: Direct manipulation and What-You-See-Is-What-You-Get (WYSIWYG) should be used to build the document. Supply a tool palette which allows the user to "tear off" new objects and drag them directly into the document.

Figure 4.6. Example sketch of a system pattern classified according to its purpose, Document Manipulator

4.2.2 Target Medium

It has already been mentioned that computers are ubiquitous in society (Chapter 2). This being the case, HCI must deal with many types of human-computer interfaces. Target media which could be covered by

patterns include:

- Applications for conventional desktop Window-Icon-Mouse-Pointer (WIMP) systems, such as a graphical word-processor or drawing tool
- Websites
- Applications for handheld computing
- Applications for mobile phones

This dimension has a similar scope to "patterns of entire systems" in the previous section. However, in this case, it is merely an attribute of the pattern collection rather than a pattern in itself. Target medium has been less of an issue with software design patterns, which devote most attention to events beneath the surface. There have, however, been a few patterns specialised for certain computing environments, e.g. parallel processing [161] and embedded systems [27].

If patterns are supposed to be timeless, applying across times and contexts, then why should a pattern language be restricted to a particular medium? Clearly, the answer lies in finding the right balance. In some sense, all patterns can be seen as belonging to one giant collection. However, to form a coherent language, the patterns need to share a similar scope. This does not mean patterns for one medium cannot draw upon knowledge from other media. Indeed, this is a great strength of the approach; patterns for newer media (e.g. handheld computers) can accelerate design quality by leveraging from existing knowledge.

4.2.3 Specialised Requirements

Some pattern languages address systems where specialised requirements are present. There are various types of specialised requirements:

Application Domain Examples of application domain include: finance, telecommunications, control. Application domains which have been covered by software design patterns include fire alarms [168] and business associations [83].

User Characteristics User characteristics can include skill level with computers, or certain needs such as systems catering for disabled users or perhaps people who work in environments with frequent interruption. "Patterns of users" were already discussed in Section 4.2.1.2. Here, though, user characteristics form a dimension of the language. Thus, we might have a pattern language for website design (the target medium), where users are blind (the user characteristic, a specialised requirement).

Software Qualities Software qualities, also called "software attributes" or "non-functional" requirements, can interact in complex ways. As discussed in Chapter 2, there is considerable tension between usability and attributes such as reliability and flexibility. The purpose of a pattern is to explore the

tensions which surround recurring problems, and describe ways to resolve these tensions. Thus, pattern languages are well-placed to address systems where usability, as one high priority, must be balanced against other high-priority software requirements.

4.3 Patterns in HCI: Existing Research

Several HCI pattern collections have been developed in recent years, in parallel to the present thesis. Table 4.1 summarises several notable contributions. In each case, the descriptions of the dimensions are somewhat simplified, and the details of each approach will be summarised further on in this section. Many languages do not explicitly state the target medium, but it can be reasonably inferred by the nature of the patterns and the supporting examples.

The first five approaches (Tidwell, Brighton Usability Group, Van Welie and Traetteberg, Coram and Lee, Wake) address mostly user-interface issues for desktop applications. Borchers' approach devotes more attention to the application domain. The next few approaches (Cybulski and Linden, Bradac and Fletcher, Perzel and Kane, Riehle and Züllighoven) look at particular types of systems, i.e. multimedia, software with forms, websites, software for manipulating artifacts. The last two approaches (Breedvelt-Schouten et al., Stimmel) relate to various systems, but feature different levels of abstraction from the other approaches: one approach considers patterns of task models, while the other considers patterns of development process.

What follows is a summary of each patterns, highlighting certain features relevant to this thesis. Given the underlying assumption of this work that tight constraints lead to collections which are more language-like in nature, particular attention is paid to the constraints of each collection as well as the relationship between component patterns. The chapters following this one present several pattern languages developed according to tight constraints, with the objective of achieving tightly-connected patterns.

4.3.1 Interaction Design Patterns

Probably the most comprehensive collection of HCI patterns is Tidwell's interaction design patterns [218, 219]. The collection divides patterns into nine categories. Each pattern can be used at various levels of scale, and the author notes that this characteristic distinguishes the collection from Alexander's language.

The patterns are generally features of the user-interface. Some are relatively low-level, such as `Chart or Graph` and `Pointer Shows Affordance`. Others are more like patterns of multiple user-interface elements, e.g. `Stack of Working Surfaces` and `Small Group of Related Things`. There are also patterns which describe the overall system, e.g. `WYSIWYG Editor` and a pattern for describing applications which have little user interaction, `Background Posture` (e.g. a printer daemon). Other patterns imply more about concepts and function, e.g. `Interaction History` and `Navigable Spaces`.

The collection is large and quite general in scope, and this could lead to patterns which form more a catalogue than a language. Tidwell does, however, use certain concepts to tie together the patterns. One

Table 4.1. Approaches to HCI patterns in the literature, with classifications according to dimensions in Section 4.2.

Approach	Level of Abstraction	Target Medium	Specialised Requirements
Tidwell [218],[219]: Interaction Design Patterns	Entire systems, Multiple and single user-interface elements, Functionality	Desktop Applications, Websites	None
Brigthon Usability Group [31]: Brighton Usability Pattern Collection	Entire systems, Multiple and single user-interface elements, Functionality	Desktop Applications	None
Van Welie and Traetteberg [223]: Amsterdam Pattern Collection	Multiple user-interface elements, Functionality	Desktop Applications, Websites	None
Coram and Lee [58]: Experiences	Multiple and single user-interface elements, Functionality	Desktop Applications	None
Wake [227]: Patterns for Interactive Applications	Multiple user-interface elements, Functionality	Desktop Applications	None
Borchers [25, 26, 24]: Interdisciplinary Design Patterns	Both High-level and low-level, Functionality	Various	Can be domain-specific
Cybulski and Linden [65]: Multimedia Patterns	Multiple user-interface elements, Functionality	Multimedia Applications	None
Bradac and Fletcher [29]: Patterns for Form Style Windows	Multiple user-interface elements	GUI Forms	None
Perzel and Kane [186]: Usability Patterns for Applications on the World Wide Web	Multiple and single user-interface elements, Functionality	Websites	None
Riehle and Züllighoven[194]: Tool Construction and Integration	Multiple user-interface elements, Functionality	Desktop Applications	Artifact Manipulation
Breedvelt-Schouten et al. [30]: Reusable Structures in Task Models	Tasks	Various	None
Stimmel [206]: Patterns for Developing Prototypes	Development process	Various	None

thread in many of the patterns hinges on the Navigable Spaces pattern. The user is seen as moving through various spaces in certain environments, e.g. on the web. This is a certain viewpoint, or underlying philosophy, since there are alternative ways to perceive of human-computer interaction in these situations, e.g. as a human-computer dialogue. This concept helps to tie together various patterns, e.g. Go Back One Step means reverting to the previous space, Interaction History refers to a list of previously-visited spaces.

4.3.2 Brighton Usability Pattern Collection

The Brighton usability patterns [31] share a similar scope to Tidwell's patterns, and most depict user-interface features. Emergency Exits, for example, urges designers to "ensure that the option to quit a program is immediately and obviously available". Show the Format Required is another good example of an HCI pattern. There are also some patterns which suggest more about functionality than user-interface. Allow Undo addresses reversibility of actions and Think Twice reminds designers to consider the consequences of undesirable actions which are unrecoverable. Some are probably closer to principles than patterns, such as Interaction Feedback — "Give visual feedback and allow the user to enable audio feedback for every interaction event". This is a statement which is still in need of patterns to indicate how it applies in various context.

The patterns form something more like a catalogue of patterns than a language. They are related to one another in their common attitude towards usability, and the patterns certainly describe the relationships at various points. This viewpoint is apparently clear to the authors themselves, as they label the patterns a "collection", not a language. It is important to note that a pattern catalogue such as the Brighton collection can certainly deliver practical benefits to designers, as Gamma et al.'s text [88] has shown.

4.3.3 Amsterdam Pattern Collection

The Amsterdam Patterns Collection [223] is another set of general-purpose HCI patterns. The patterns are similar to those in the previously-mentioned collections. For example, Navigating between Spaces resembles Tidwell's Navigable Spaces, and Managing Favourites, resembles Tidwell's Bookmarks.

Van Welie has explained that the new collection was begun because he "wanted a collection of patterns that is strictly focused on problems of the end-user and not problems of designers" [224]. This fits with the apparent objectives of the previously-discussed collections. The collection makes an interesting contribution with respect to the format and creation process. The patterns are represented in XML and the intention is that pattern authors can submit their patterns for inclusion in the website. This concept may lead to less redundancy among pattern languages and provide a common repository for people to search. At the same time, there is also a risk that the benefits of the *language* concept will be lost. It may be possible to somehow link patterns such as Tidwell's and the Brighton Groups's together. At this stage, it is unclear

how more tightly-constrained languages with closely-coupled patterns could fit into a universal repository.

4.3.4 Experiences — A Pattern Language for User Interface Design

Experiences [58] was probably the first attempt at a general-purpose pattern language for HCI. As with Tidwell's collection, the patterns are mostly based on the user-interface and are quite general in scope. Where Tidwell has deliberately avoided creating a map of the patterns, Coram and Lee are able to do so, because the patterns are designed to flow, at least approximately, from high-level to low-level. Thus, the initial pattern, `Interaction Style`, is a "meta-pattern" urging developers to choose a style appropriate for the user base, and suggesting four patterns of interaction style. These are: `Entry Form`, `Selection Menu`, `Conversational Text`, and `Explorable Interface`. These interaction styles lead towards low-level patterns covering specific user-interface elements.

4.3.5 Patterns for Interactive Applications

Wake's patterns for interactive applications [227] extend two patterns described by Beck [15]. Some patterns are quite isolated (e.g. `Existing Extension Language` and `Computer Magic`, which suggests computers add value to users, rather than a pattern). Several patterns form instructions for placement of user-interface objects. For instance, `Zones of Similar Persistence` says "Partition the window into zones. Items in a given zone should change at about the same rate." and `Space Proportional to Importance` says "Make the space for a zone proportional to the importance of the objects and relations in that zone". There is some coupling among the patterns by way of the structuring metaphor, i.e. the interface contains zones of related information. However, patterns in this language seem somewhat similar to design principles, and it would be probably more useful if there were patterns for particular configurations which are able to solve particular problems. For example, maybe small interfaces (e.g. mobile phones) require fewer zones and this may constrain solutions in certain recurring ways.

4.3.6 Interdisciplinary Design Patterns

Borchers [25, 26, 24] is interested in preserving the original goals of pattern languages, while still adapting Alexander's concepts where necessary. An important contribution is the emphasis on pattern languages as a facilitator of interdisciplinary work, encompassing interaction between HCI specialists, software engineers, users, and domain experts. He has observed that Alexander's patterns are understandable not only by architects but by lay-people and contrasts this with the technical nature of software design patterns, whose class diagrams and sample code are incomprehensible to end-users [26].

To facilitate communication across disciplines, Borchers documents HCI patterns in accessible language supplemented by visual aids, similar to Alexander's introductory photographs and simple hand-drawn sketches. In addition, he proposes that an HCI pattern language can be combined with an applica-

tion domain pattern language and a software engineering pattern language. The HCI-SE-Domain triad is expected to weave together to yield a smooth integration among the disciplines.

A musical exhibit has been used to demonstrate this approach. Here, a music pattern language is necessary to capture the domain. One such pattern is `Triplet Groove`, sketched in Figure 4.7 [25]. Patterns like this are explicit to users while they interact with the exhibit. Since the triplet groove has certain parameters, users can tweak the parameters and hear the effect. An HCI pattern used for the exhibit is `Incremental Revealing`, the pattern of revealing functionality in an item only after the user has shown initial interest. There are also software design patterns. `Transformer Chain` sees several modules transform a stream of data, one after another, and has obvious applications to processing of musical data.

Pattern 4-7. Triplet Groove

Context: Playing music in the `Jazz Style`.

Forces:
- Players need to create a swinging feeling that the straight rhythm from other musical styles does not convey.
- But sheet music cannot include all rhythmic variances; it would become unreadable.

Solution: Where the score contains an evenly spaced pattern of eighth notes, shift every second eighth note backwards in time by about one third of its length, shorten it accordingly, and make the preceeding eighth note one third longer. ...

Figure 4.7. Borchers' musical domain pattern, Triplet Groove (based on Borchers, 1999 [25].

4.3.7 Multimedia Patterns

Unlike the approaches above, Cybulski and Linden's patterns [65] explicitly address a well-defined subset of applications, namely multimedia applications. Multimedia applications are far from homogeneous, but designers are likely to come across many similar challenges when working within this category. Furthermore, the language concerns reuse of artifacts within multimedia application design, further tightening the scope. This leads to the question of whether patterns constrained in scope might exhibit greater coherence.

Although their publication [65] contains only five of the many patterns they envision, the patterns do provide a glimpse of the sense of connectedness which might be offered by a tightly-constrained language. There is a pattern called `Glue` which suggests sticking several artifacts together into a newly-formed composite object, e.g. a group of words in a word-processing document, or an image containing several shapes. Another pattern, `Define and Run Presentation`, solves the problem of creating several channels, each showing a different sequence of artifacts. Part of the solution is to apply a particular kind of `Glue`, a `Partial Order Glue`. In fact, most patterns use `Glue` in some way. This is a demonstration of the strong

inter-relationship which is possible among patterns. Because a common technique in these applications is sticking several artifacts together, i.e. applying `Glue`, many of the other patterns share a common feature. This makes the patterns easier to learn, since they build on existing knowledge, and also easier to apply, since they all push design in the same direction.

4.3.8 Patterns for Form Style Windows

Instead of considering an entire target medium, such as WIMP or multimedia, Bradac et al.'s patterns [29] are constrained to one particular user-interface feature: the form-style window. This could constitute the entire application, but is more often one part of the overall application.

Five patterns reside in the collection. The first pattern, `Subform`, suggests breaking a form into subforms. This is a good example of a straightforward prescriptive pattern, and it forms the groundwork for the rest of the language. The other patterns provide guidance on the decomposition of the form, and the dynamic communication mechanisms between the various components. `Alternative Subforms` suggests using state data to produce an appropriate subform. For example, a user who selects Home Country of USA needs a particular address format, while a user who selects Australia needs a different format. The address subform depends on the Home Country field. But this opens up a new problem: what if the user alters Home Country? The `Subform Selection` pattern shows how to handle it with a polling mechanism. Subsequent patterns offer further resolution.

As with Cybulski and Linden's language, there is a tight scope involved in the patterns of Bradac et al. [29]. Once again, there is evidence of a well-defined, coherent, set of patterns which demonstrates the notion of a pattern *language*. A designer can approach the language with a very specific goal in mind: to design a form-style window. The patterns then take the designer through the various decisions which must be made.

4.3.9 Usability Patterns for Applications on the World Wide Web

Perzel and Kane's contribution [186] constrains the target medium to websites. Some patterns describe the user-interface, for instance `Required Field Markers` to show users which fields are compulsory and `Plan B` to provide alternative representations for users with little or no graphic capability. Other patterns are "system-based", i.e. define functionality, organisational practice, or other conceptual issues. Among these are `Server-Side Validation` and `Searching the Web`.

The tight scope might lead to the prediction of a well-connected language. In fact, the collection seems somewhere between a catalogue and a language. There is a Related Patterns field which does provide guidance at various points. For instance, `Policy Statement` suggests that the policy statement should be prominent; therefore, it directs designers to `What they See is All They Get`, which suggests critical elements should be placed near the top of the page. This type of relationship is present in several patterns;

it can help to form a prescriptive design process, but does not really convey an overall design philosophy. The most likely reason is that websites, as a broad category, may not be an appropriate target medium to form a well-integrated language. The collection of web patterns is useful, in the same way as Gamma et al.'s patterns are an excellent tool for software designers, but the language aspect is not captured as well as it might be if the scope was limited to, say, e-commerce portals or community forums. Perhaps a general lesson is that although constraints help to produce a smoother relationship among patterns, as demonstrated by Bradac et al.'s work, certain constraints may not be amenable to the formation of a pattern language.

4.3.10 Tools and Materials

Riehle and Züllighoven's language [194] begins not with patterns but with "Design Metaphors". These form a set of concepts which effectively define an ontology on which the subsequent patterns are based. The design metaphors are based on the authors' "leitmotif of skilled experts whose work we wish to support with the appropriate software".

The metaphors are `Materials`, `Tools`, `Aspects`, `Environment`. A `Material` is a work artifact that users manipulate. A `Tool` is an object which can facilitate the manipulation. An `Aspect` is an interface to the material which defines what is required in a tool. For instance, a set of appointments has two aspects: it is Listable and Editable, hence it must be manipulated by tools capable of listing and editing. The `Environment` serves as a space where tools and materials are organised and utilised. These metaphors are derived from a particular worldview held by the authors. They state the following (p.10):

- *People have the necessary competence and skills for their work.*
- *There is no need to define a fixed work flow, because people know what they do and can cope adequately with changing situations.*
- *People should be able to decide on their own to organize their work and their environment (including software they use) according to their tasks.*

As such, there is a smooth transition from the authors' underlying philosophy to the design metaphors. There is also a smooth transition from the metaphors to the design patterns, which are software architecture patterns in the style of Gamma et al.'s [88]. They show how to implement systems which support the metaphors. For example, `Tools and Materials Coupling` makes a very concrete statement: "each aspect becomes a class interface in its own right, called an aspect class". Thus, there is an aspect class Listable, which defines what a List tool must accomplish. The class for a set of appointments inherits from Listable. Several design patterns follow in the same spirit. They address tool composition, tool presentation, presentation synchronisation, and flexibility of tool functionality and presentation.

The strength of this language is in its strong sense of individual character. The authors do not just "pluck" patterns from a wide range of related systems. They begin with a particular worldview, and use

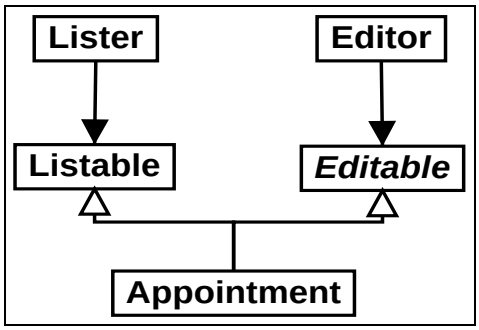


Figure 4.8. Example from Riehle and Züllighoven's Tools and Materials language (based on Riehle and Züllighoven[194]). Lister acts on anything which is Listable. By making Appointment implement the Listable interface, the designer is enabling Lister to operate on Appointment.

this to form a basic ontology. They can then build on this ontology to provide a language of patterns, made coherent by their mutual adherence to the underlying philosophy.

4.3.11 Reusable Task Models

A different level of abstraction is suggested by Breedvelt-Schouten et al. [30]. Instead of looking at user-interface elements or direct functionality, the authors suggest a kind of pattern based on another key construct in HCI: user tasks. Although the authors do acknowledge a proximity to design patterns work and applications, they do not discuss patterns in quite the same way, i.e. in a formal notation such as Alexandrian form. This may simply be because they are offering an initial proposal at present [30]. Nevertheless, the reusable task models is clearly a form of HCI patterns work. It has also been embodied in the "Generic Tasks" concept described in Section 4.4.4.

Breedvelt-Schouten et al. call the patterns "task templates". Based on their experience creating task models, it is asserted that several templates continue to emerge. The simplest template is `Multi-Value Input`. As Figure 4.9 shows, the task requires the user to Edit Values, then Send Input. Edit values itself can be broken down to editing attribute 1, attribute 2, and so on, simultaneously and through several iterations if necessary. The template is exemplified in a sales processing application, where the overall `Multi-Value Input` task is instantiated with a specific task, `Handle One Order`, and the other tasks are similarly mapped to application-specific tasks.

This form of reuse has high face validity; it is obvious that tasks do recur in this way, and it would save modellers effort if they could be captured for reuse. It is also evident that the patterns could be bound by an overall set of principles. This can be seen in the broad sense of interaction paradigms. GUI systems tend to be structured around "Object-Action" tasks, while command-line systems tend to be structured around "Action-Object" tasks. Thus, the `Multi-Value Input` task makes sense for a GUI system because

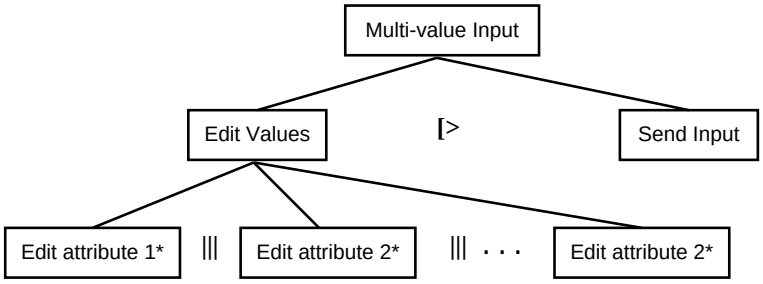


Figure 4.9. Example from Breedvelt-Schouten et al.'s task pattern for Multi-Value Edit (based on [30]). Reusable task specification using ConcurTaskTrees notation. Multi-value Input consists of Edit Values followed by Send Input. Edit Values consists of editing each attribute, in any order, as many times as desired.

it entails specification of the object (via its attributes) before the action (i.e. send input). A command-line task would reverse the ordering. Therefore, patterns of this nature would work best for a constrained set of systems sharing common principles for task structuring. In this context, they would offer valuable assistance to designers who are trying to understand how particular tasks can be implemented according to the common expectations of users.

4.3.12 Patterns for Developing Prototypes

Stimmel's collection [206] deals with prototyping activity, a development process related to HCI rather than an HCI construct such as a task model or a user-interface design. The first pattern establishes a common ground for the remainder of the patterns. The pattern, `Use It and Lose It` proposes the prototype should be discarded upon client acceptance. The other patterns build on this advice. `Engage the Client Early` advises developers to rapidly generate prototypes which moderate requirements discussions, capitalising on the fact that the prototype does not need highly-quality code (code quality is obviously compromised by rapid prototyping; see Chapter 2). `Feature-Free Zone` encourages prototypers to focus on the core requirements (and implicitly implies prototypers should avoid those fancy features which may be present in the final system).

An important aspect of the language is the sense that it establishes an underlying philosophy and explains how to conform to it. It would be equally feasible to devise a pattern language around a well-managed evolution from prototype. However, this would lead to an entirely different set of patterns. `Feature-Free Zone` would often be uneconomical in this context, because unanticipated peripheral features may require expensive changes to the core. As with the Tools and Materials approach, the patterns in Stimmel's language are therefore bound by a common viewpoint, a property which that should lead to a

coherent development process.

4.4 Related Research

The work discussed in the previous section represents direct attempts at capturing patterns for HCI. In all cases, the authors have produced collections of patterns which have usability as a primary goal. It has also been instructive to survey the literature for approaches which are related, less directly to HCI patterns. Research along these lines has also had an influence on the thesis, hence several efforts are outlined below.

4.4.1 Construction-Driven Design

Casaday has suggested there are two ways to drive design: by evaluation or by construction [38]. The former technique sees designers producing work haphazardly, evaluating, and reworking. The latter method, which corresponds to a design patterns approach, "relies on forms or procedures and uses evaluation only for selection and tuning"(p.358) [38]. In more general terms, construction-driven design might be seen as a process driven by tools which enable the design process to move in a forward direction from the problem statement or requirements document. Among the benefits of a construction-driven approach are that it is likely to produce faster results, transfer more easily, and facilitate communication for a design team.

Aiming for a generative mode of design, Casaday employs templates. A template is essentially a form filled out by a designer. The templates focus developers' attention on user-centered issues rather than on the final system. The final system is designed after the templates are filled out, hence the process is more sequential than an evaluation-driven approach. An example of a template is a matrix for training requirements. The designer indicates what level of training will be required for (a) operating the system, (b) applying the system to particular tasks, (c) understanding the task itself. Each question can have zero or more standard answers checked, i.e. expert help, self-tutorial, etc.

This is a reusable design technique with low-technology tool support, rather than a reusable knowledge base like patterns have to offer. It is certainly not the only reusable design technique in existence. What is important about Casaday's work for the present thesis is that it offers an articulation of the benefits offered by a construction-driven design process, a process which can be facilitated by patterns.

4.4.2 Applying Existing Software Patterns to HCI

Gamma et al.'s well-known software design patterns [88] have been adapted to an HCI context by Noble [174]. He "translated" six patterns to GUI counterparts, explaining that although the patterns do not directly address usability, they do reflect patterns in interfaces designed for usability. An example is the `Singleton` pattern. In software design, this is the situation where a class has exactly one instantiation. A `GUI Singleton` is a GUI object with exactly one instance, e.g. the Recycle Bin in MS-Windows 95 cannot

be duplicated nor deleted.

Noble offers the patterns as an experiment. The direct benefit to developers is probably small, but the patterns could act as an input for patterns which are purpose-built for HCI reuse. Adapting patterns in this way also has the benefit of preventing lessons in one discipline to be reinvented by another, a consequence of the timeless quality of patterns.

4.4.3 End-User Awareness of Patterns

The view of patterns as a "Lingua Franca" to facilitate interdisciplinary work has already been discussed, in relation to Erickson [75] (Section 3.3) and Borchers [26] (Section 4.3.6). The common vocabulary of HCI patterns is mostly discussed as a means of sharing design ideas among HCI specialists, software engineers, managers. End-user representatives may also be included, but they are only proxies for the ultimate users. A stronger form of application patterns in HCI is the notion of end-users being *aware* of the patterns and interacting with them.

This situation is demonstrated to a degree by Borchers' interactive music patterns [26]. It is taken to much greater lengths by Harris and Henderson [102]. The 2020 AD scenario they envisage is encapsulated by the following scenario (p.14) [102]:

> *Olivia needs to track down the source of the problem. She asks the office to show her the Heavy Metal [Olivia's company] operational patterns that strongly match her refined definition of the problem. She quickly gets a display similar to the earlier "sea of nodes" but this time organized by strength of match to her definition. She moves the two most similar patterns on top of each other, and the system shows her discomfort markings where they differ. Olivia marks some of the elements of the pattern as "don't care" and adds a variant sub-pattern in one place, and the two patterns merge, with the "don't care" sections grayed out. The change ripples through the sea, allowing some patterns to merge and moving many of them closer together.*

The view is that organisations use patterns in much the same way as guiding principles are used now, except the patterns by their nature provide much more concrete advice about how problems should be addressed. The patterns provide a way for the organisation to share a common focus and the process of using and improving the patterns, combined with appropriate tools, allows organisations to provide value to their customers in a reusable manner.

4.4.4 Generalising HCI Claims and Requirements Reuse

Sutcliffe and Carroll have discussed how claims in HCI can be generalised to improve reuse of HCI knowledge [212]. They propose the use of "generalised claims", design tools similar to patterns. A Generalised Claim, like a design pattern, is a description consisting of several fields, intended to help designers reuse

successful design strategies. The claim is a generalised statement about how users react to a certain feature or concept. Thus, the concepts are very similar, although claims are somewhat more focused on cataloguing HCI knowledge in a medium which can be shared and evolved by researchers and practitioners. Patterns, in contrast, tend to take existing theory and package it in a manner which is optimal for practical design use.

A generalised claim example is `Color-coded Telegraphic Display` which suggests that in educational systems, certain benefits are derived from providing a colour-coded list of goals. Fields of a generalised claim include: Artefact (something that exemplifies the feature described by the claim), Description, Upside, Downside, Scenario, Effect, Dependencies (activities which must be conducted to implement the feature), Issues (similar to keywords), and Scope.

The two concepts both share the goal of capturing theoretical knowledge in ways which will be of practical assistance to designers. Some recent work by Sutcliffe and Dimitrova[213] suggests one way they might interact with each other. To reusable claims, the authors add a Pattern field. The Artefact field is retained from the previous schema, and is now an example of the Pattern. Thus, the Pattern's role is to summarise the set of artifacts which use the claim. The approach could lead to a repository of "Patterns+Claims", in which a large number of claims could be documented across HCI. This is similar to the repository of patterns suggested by Van Welie (Section 4.3.3).

A related work is Sutcliffe and Maiden's domain theory for requirements engineering [214]. In this theory, it is asserted that "knowledge can be reused between dissimilar domains ... if an appropriate abstract model (e.g. loaning resources) can be transferred" (p.174) [214]. Thus, the authors have provided a notation and process for extracting such models. Meta-domains can form the working area of HCI claims as described by Sutcliffe and Carroll [212]. Also related to the domain theory is what they refer to as "generic tasks". These are tasks which operate on objects or agents, either in grounded or meta-domains. They abstract from specific tasks. An example is diagnosis: "determining the cause of some malfunction in a system, locating its cause and proposing remedial treatment", which maps to medical diagnosis or fault-finding in an electrical system. Generic tasks are similar to Breedvelt-Schouten et al.'s task patterns, and the concept is pursued further in Chapter 8 (my own notion of generic tasks, which differs somewhat, is also discussed there). If particular domains and generic tasks are well-understood, it should be possible to capture and document reusable knowledge indicating how designers can support users in these contexts. This is certainly achieved in the case of generalised claims, and there is no reason why it cannot be transferred to a more direct application of patterns.

4.4.5 Pattern-Centered Development Process

Section 3.2.3.5 commented that one of the challenges for real-life developers is effectively integrating patterns into a coherent development process. Granlund and LaFrenière's "A Pattern-Supported Approach to the User Interface Design Process" [93] represents an attempt to improve the situation.

In a manner similar to the classification in this chapter, several different pattern types are defined.

- *Business Domain Patterns* such as `Public Healthcare` provide a label for business domains and document the typical forces involved in those industries.
- *Business Process Patterns* such as `Healthcare on Demand` capture the processes and actors which deliver the goals of the business domain patterns.
- *Task Patterns* such as `Patient's Record Review` document tasks typically conducted by actors adhering to business processes.
- *Conceptual Design Patterns* such as `Question Response` describe human-computer dialogue, functionality, data, and other aspects of the software which help users achieve their tasks.
- *Design Patterns* such as `Timeline` are patterns describing elements in the user-interface.

Each level of pattern delegates details to patterns which reside at the next level of detail. This relationship enables a linear development process. Business patterns contribute to system definition. The system definition combines with task patterns to produce a task/user analysis. Conceptual design and design follow likewise.

4.4.6 Patterns of Organisations

Coplien has captured patterns of organisations with the intention of facilitating generative design of organisational process [54]. In his pattern language for software development organisations, numerous patterns describe what types of structures are appropriate, depending upon technical and non-technical factors. For example, `Fire Walls` suggests that a management role is required to shield developers from uninvited outside comments. The complex relationships among individuals within an organisation leads to a rich set of patterns, with strong inter-pattern relationships. `Fire Walls` leaves the problem that potentially valuable outside information is lost. It is balanced by a second pattern, `Gatekeeper`, which argues in favour of a communicative technical person to disseminate external information to the development team.

A similar language has been offered by Cockburn [42]. A number of patterns in this language are detailed software patterns. But in all cases, the rationale is largely driven by social considerations, a consequence of the viewpoint that social and technical issues intertwine. `Facade`, based on Gamma et al.'s [88] pattern of the same name, demonstrates this dual perspective. The pattern suggests that keystroke validation should occur at the GUI level, stating "Provide a single point of call to volatile interfaces". It fits with a social view of development because it allows for a useful division between those coding behind the facade and those external to it. It therefore supports earlier patterns in the language such as `Variation Behind Interface` and `Subsystem by Skill`.

Organisational patterns like Stimmel's [206] prototyping language (Section 4.3.12) provide advice on development processes which improve HCI, and therefore have a special relevance to HCI. But even without HCI consideration, organisational patterns languages have been a source of inspiration to the present work. They are concerned with human issues more than their software design counterparts. The affinity to the present work is particularly strong in the case of patterns for software development processes, exemplified by Coplien and Cockburn. These languages weave between the "hard" details of software construction and the "soft" details of social interaction among development staff. This thesis weaves between similarly detailed aspects of software construction and the less well-defined principles of HCI.

4.4.7 Detailed User-Interface Design Patterns

There have been several independent efforts to capture user-interface patterns. In these cases, the focus is on the technical aspects of the user-interface, attributes like reliability and flexibility, rather than usability. These non-functional attributes can indirectly assist usability, e.g. a more flexible system can be more responsive to users' change requests. However, the patterns are not intended to encapsulate usability principles.

Buschmann et al. [34] have described the well-known Model-View-Controller (MVC) architectural style [130] in a pattern format. Hussey and Carrington [112] have also described several patterns, some from Gamma et al. [88], and shown how they can be used to create either a Model-View-Controller architecture or a Presentation-Abstraction-Controller [63] architecture. Coldewey has described a pattern language for user-interface software. `User Interface Layer` is the initial pattern which sets the scene for the rest of the language, suggesting a split between the user-interface subsystem and the domain-level subsystem (along the lines of Seeheim [62] or MVC). Gamma et al. frequently draw inspiration from user-interface tools and frameworks which they have worked on (e.g. Interviews, ET++, HotDraw).

These languages are of interest because, even though usability encompasses concepts of tasks and usage context and user characteristics, there is an inevitable association between usability and user-interfaces. It is unrealistic to expect user-interface appearance to be completely shielded from detailed design work. User-interface code inevitably impacts on what the user sees and often on the dialogue between human and computer agents. It would therefore be useful if patterns for user-interface design incorporated usability principles into their underlying principles.

4.5 The Present Thesis

4.5.1 Focusing on the "Language" Aspect

In respect to the patterns concept, this thesis is primarily concerned with the notion of *language*. This interest arises from the apparent benefits languages offer, discussed in Section 3.2.3.1. This, coupled with

the observation that many approaches to patterns in HCI and SE stop short of pursuing a pattern *language*, has led me to explore the idea of pattern languages for HCI.

In aiming for a pattern language, it is apparent that the most effective approach is to constrain the language's scope. A good example of a tightly-scoped, well-integrated language, is Molin and Ohlsson's pattern language for fire alarm systems. The pattern relationships make it clear that the authors are drawing on examples from related systems. The first pattern is `Deviation`, which suggests representing a deviation from normal state as a Deviation object, with subclasses such as `Alarm` and `Disturbance`. A `Point` pattern suggests using a Point class to interface with sensors and alarms. There is a strong relationship between these patterns, because Input Points (sensors) lead to Deviations and Deviations lead to Output Points (alarms). The language proceeds to describe other related concepts, e.g. pools of deviations for common access and scheduling of events involving dynamic behaviour of the various components. Although the language is small and not applicable to a broad range of systems, it is easy to see how it could be readily applied in those situations where it does apply, i.e. fire alarm design, and could also educate designers in related fields, such as other hazard detection systems.

The benefit of a tight scope is also apparent in the HCI patterns examined in this chapter. Those with characteristics associated with pattern languages tend to have a tight scope. This is logical; a tight scope implies a more homogeneous set of example applications from which patterns are derived. More homogeneous applications will share similar design features, hence there is a high likelihood of finding several examples of the design features working with each other. And finding examples of common design features working with each other is a major requirement for capturing a pattern language; it is the close relationship between patterns that produces a coherent set of patterns.

In contrast to the fire alarm example, a broad scope leads to a whole array of design features, with many sharing little in common other than perhaps being at the same level of abstraction. Consequently, a larger and less coherent set of patterns will emerge, and might be labelled a "pattern catalogue" instead of a pattern language.

This is not to say a catalogue has no benefit. Indeed, it is much easier to fill a catalogue with many patterns than complete a language of the same size. This ease of construction may help to produce a tool which is more comprehensive and more capable of responding to changes in technology. As long as effective information retrieval techniques are adopted, a large HCI catalogue may be a valuable supplement to high-level HCI principles. It will have broader applicability and a greater number of potential contributors.

Pattern *languages* for HCI, on the other hand, may take longer to reach widespread acceptance. They require more effort to construct, since each pattern must be consistent with the others, and all must work towards a common set of principles. If they address situations with limited scope, they will not apply to as many developers. However, it would be naïve to set out creating a general-purpose pattern language with so few examples of coherent HCI pattern languages. That may happen in the future, but a more overwhelming concern at present is to ask whether we can generate coherent pattern languages even for

Table 4.2. Approaches to HCI patterns taken in the present thesis

Approach	Level of Abstraction	Target Medium	Specialised Requirements
Planet	Development process, multiple user-interface elements, functionality	Desktop applications, Websites	Users from heterogeneous cultural backgrounds
Safety-Usability Patterns	Task, functionality, multiple user-interface elements	Desktop applications, Embedded software	Safety-critical systems
MMVC	Detailed software design	Desktop applications based on MVC	Specific set of generic tasks

tightly-constrained families of systems.

4.5.2 Pattern Languages in this Thesis

The present thesis contains three pattern languages, all with different types of constraints:

The Planet Pattern Language for Software Internationalisation (Chapters 5, 11) A language to assist in the development of software for international users. The language addresses the specialised usability forces which arise in accommodating cultural differences, initially at a high level (Section 5.3). It is revisited towards the end of the thesis (Section 11.3) in a manner which weaves the high-level considerations with low-level design issues.

Safety-Usability Patterns (Chapter 7) A language to support usability in safety-critical systems. The patterns are divided into four groups: patterns of task structure, patterns of input mechanisms, patterns of information presentation, and patterns of machine control.

Multiple-Model-View-Controller (MMVC) Patterns (Chapters 8, 9, 10) Detailed design patterns for development according to the MVC architectural style. The patterns show how recurring user activities, such as creating an object, can be captured at the detailed design level. MMVC is a proof-of-concept rather than a complete language. It combines with a code-generation tool and a set of reusable classes to form the MMVC framework.

Table 4.2 summarises where the work in this thesis fits in to the classification provided earlier. Compared to most other HCI patterns approaches, each of the languages described here has significant constraints. In Planet, it is the user base. In Safety-Usability patterns, it is the safety-critical constraint. In MMVC, it is the underlying MVC architecture, focusing particularly on implementing a small, defined, set of tasks. These constraints provide a way to examine whether tight constraints can produce tightly-integrated pattern languages.

A set of general-purpose usability patterns can realistically cover only one level of abstraction. It can still probably be reasonably well-integrated if it does this, but only if it is constrained to a certain target medium, in which a well-defined set of principles exist. The hypothetical example in the last chapter of a pattern language for Macintosh developers demonstrates this point (Section 3.2.3.4). However, it is more interesting to look at pattern languages which cover a range of abstraction levels. Such languages are more useful for interdisciplinary design. In constraining Planet and safety-usability patterns to particular specialised requirements and target media, it has been possible to expand in level of abstraction. Planet in particular crosses boundaries between organisational process, high-level usability design, and detailed software design. This is possible because it only deals with a single aspect of the software. The safety-usability patterns are less global in scope, but nevertheless cover several different aspects of interactive design.

This thesis is about reuse, an area encompassing more than just patterns or pattern languages. In addition to the pattern languages described above, the thesis also explores reusable code and software tool support. Planet is based on the concept of an online repository, a broadly-accessible knowledge base containing information about users. A prototype of the repository has been developed and is described in Chapter 6. MMVC contains a number of reusable software classes. In addition, it contains a tool which can be used to generate code for certain classes. Thus, MMVC is labelled a “framework” rather than a pattern language.

4.5.3 The Process of Pattern Discovery

Section 3.2.3.2 identified three approaches to pattern discovery. In this thesis, the approach may be broadly categorised as artifactual in approach, i.e. the patterns are produced by surveying and analysing existing systems. This was preferable to an introspective approach, because it allowed many diverse systems to be studied. Similarly, a sociological approach would require substantial coverage of specific systems, and the breadth of systems would be reduced. Furthermore, this type of approach was ruled out because it has apparently not been applied before, for any type of pattern language.

In following an artifactual approach, the basic steps were as follows:

- Identify language scope and design principles.
- Study systems within language scope, identifying features which conform to design principles.
- Identify clusters of similar features, i.e. situations in different applications where the same basic problem has occured, and has been resolved by the same basic solution.
- Create a pattern for each cluster.
- Study how the patterns relate to each other, by looking at the pattern definition as well as the relationship among example features in each cluster.

The capture of appropriate features relies on expert judgement, and it is acknowledged that expert judgement has its flaws. This is well-known in the field of usability testing, where user testing often reveals errors that experts had not anticipated, and experts complain about issues which may have little practical impact (seeSection 2.3.1.2).

Despite these flaws, expert evaluation seems to be the optimum approach for research of this nature. User testing, aside from resource issues, would be undesirable because we are focusing on specific features, whereas users would be exposed to the entire system. Expert judgement would not be eliminated, since only expert judgement could determine how user results related to particular features.

Given that expert evaluation is the most appropriate path for assessment of candidate features, it is still important to ask whether anything can be done to improve validity. In the case of expert evaluation, a set of guidelines, or principles, is used to structure the process. Here, too, the language's principles can be used to ensure features are evaluated in a more objective manner. Thus, the principles guide the evaluation process as well as indicating the type of system which will be built when using the pattern language.

4.5.4 Presentation of Patterns used in this Thesis

Section 3.2.3.3 discusses how patterns may be presented. The overall form of patterns here follows Gamma et al.'s approach, in which the field-description pairs are explicit, i.e. each field ("Problem", "Forces" etc.) forms a title for the description. This was deemed preferable to the more prose-like format of Alexander and others, as it seems more conducive to being used as a reference, answering a designer's query to a specific problem.

As for the specific fields, the fields identified as mandatory by Mezaros and Doble [164] are all present — Name, Context, Problem, Forces, Solution. To improve usability, the Problem and Solution begin with a boldfaced summary statement, which is usually followed by more detailed elaboration.

In addition, the following optional fields are always used in this thesis:

Resulting Context This is critical to the coherence of a language, as it allows a pattern to identify other patterns that may solve a problem which arises in the process of applying this pattern.

Examples Explicit examples are an excellent way to make the patterns practical as a reference tool. They also illustrate the pattern's description and can help the reader assess the pattern's validity. The reader will be able to judge a pattern's validity based on how much the reader feels the examples relate to the pattern's core problem, and to what extent they resolve it in a manner which optimises the pattern langauge's guiding principles.

The following optional fields are sometimes used in this thesis:

Rationale This is used only when the rationale is not apparent from the problem, forces, and solution.

Image Images were initially unused as it was felt the text would provide adequate coverage. However, the study in Chapter 7 revealed that images were actually quite important. Furthermore, emerging HCI pattern languages were using images to good effect. These factors suggested that images should be adopted, and they were consequently used in the detailed Planet patterns.

Design Issues This was used in the Safety-Usability patterns, to cover more detailed guidelines than in the Solution. In the case of the `Warning` pattern, this field comments on styles of auditory alerts. The other pattern languages did not cover this depth, hence this type of information was covered only in the core solution field for those languages.

The following optional fields are never used in this thesis:

Related Patterns, Aliases Related patterns in the same language are covered in Resulting Context. Related patterns in other languages, and aliases, are adequately addressed as part of the elaboration of the solution, in the Solution field.

Code Samples The detailed design patterns incorporate code samples, where appropriate, as part of the Examples field.

Acknowledgements References to related work are covered wherever the related work is cited. Other information is covered in the Acknowledgements section of a paper or this thesis.

Validity Ranking This field is quite rare in pattern languages, and it was decided not to use it here because it is highly subjective. It is also a complex issue, given that a pattern's validity might depend on the type of problem that the reader is facing. It seems more appropriate for a reader to appraise the pattern based on the reader's own skills, and this is where the examples are important in providing a case for the pattern's validity.

4.6 Discussion

This chapter has presented a classification scheme for pattern collections which consists of three dimensions: level of abstraction, target medium, and specialised requirements. The scheme has been used to classify twelve approaches to patterns which have arisen in the past few years. It is apparent that some collections are closer to the concept of a "language" than others, as demonstrated by the inter-connectedness among the patterns. A tighter scope offers the possibility of tighter relationships among the patterns.

This thesis investigates what is meant by a pattern language for HCI. To achieve a tight pattern language, it was therefore reasoned that tight constraints should be applied. This has been achieved with Safety-Usability Patterns, with their focus on safety-critical systems; MMVC, whose patterns focus on a small set of tasks implemented in the MVC architecture; and Planet, which focuses on internationalised systems.

In the next part of the thesis, High-Level Design Reuse, high-level reuse is explored via the Planet pattern language; the concept of a user knowledge repository introduced by Planet; and the Safety-Usability Pattern Language. The third part, Detailed Design Reuse, explores the inter-weaving of usability and software design. This is explored with the creation of the MMVC framework and further demonstrated by revisiting Planet to incorporate detailed design considerations that are specific to the types of problems Planet is addressing.

Part II

High-Level Design Reuse

Chapter 5

A High-Level Pattern Language for Software Internationalisation

5.1 Introduction

As discussed earlier, the most appropriate way to study the language aspect of patterns is to choose a tight scope in which the patterns apply. A pattern *language* should exhibit connections between patterns which are so strong that the patterns may be referred to as being *inter-dependent* on one another. Focusing on a specific area helps achieve this purpose by forcing the language author to draw from examples in systems with similar purposes.

To provide a tight scope, the work presented here focused on software internationalisation. Users from different cultures face particular challenges which must be accommodated in software design. Several factors make internationalisation a suitable topic for a pattern language. There is a reasonable amount of theory in existence. More importantly, since pattern languages aim to capture real-life examples, there are many existing internationalised systems which can be studied. The constraint also fits well with this thesis because the concerns range from high-level issues such as cultural psychology down to low-level details like databases of text strings.

As a further means of establishing a tight scope, the target medium is largely limited to conventional desktops, with some discussion of websites also included. Websites are worth studying because they are abundant and easily accessible, with internationalisation issues similar to desktop systems. The level of abstraction varies from high-level to low-level, and includes not only software but also organisational process. The key constraint is nevertheless tight enough to capture highly-connected patterns. Since the same systems can be studied from both high-level and low-level perspectives, interesting relationships can emerge concerning the relationship between the two levels.

This chapter covers the high-level patterns in Planet, those concerning culture modelling, design of

system features, and user configuration of these features. Following the first aim of this thesis, the purpose is to demonstrate how an integrated, high-level language can support usability. Chapter 11 adds low-level patterns to the language, patterns which relate to the detailed software design of systems which exemplify the high-level patterns. The present chapter begins by outlining the methodology for pattern discovery, in Section 5.2. Section 5.3 is a background on the key constraint of the language, software internationalisation. The patterns themselves are contained in Section 5.4. Section 5.5 is a discussion of Planet and its implications for HCI patterns.

5.2 Methodology for Pattern Discovery

It is worthwhile elaborating on the general approach for pattern discovery mentioned in Section 3.2.3.2, showing how the process was conducted for this specific pattern language.

The patterns were not discovered by a formal process; however, it is possible to identify certain activities performed while developing the patterns. The activities were performed iteratively, although some activities (such as developing background knowledge) occurred mostly at the start whereas others (such as pattern documentation) occurred closer to the end.

Study Internationalisation Issues The first activity began with a detailed analysis of requirements for international users. Section 5.3 summarises the results of that research.

Identify Implications of Analysis It is not enough to understand the needs of international users; implications of the research should be made explicit. Thus, several recommendations and implications for practitioners and researchers were suggested as a result of the background study on internationalisation.

Document Principles Since pattern languages demonstrate how principles are achieved, it was useful to make explicit the principles behind the patterns.

Capture and Document Patterns Based on the previous analysis, as well as the internationalised systems which were studied during that time, patterns were captured, complete with illustrative examples.

Document Pattern Relationships As the patterns were documented, relationships among the patterns began to surface. Sometimes, writing a pattern led to a new, related, pattern being captured.

Review Patterns The patterns were subject to much review by myself, my supervisors, and Todd Coram, the shepherd for the Pattern Languages of Program Design conference [149]. Following reactions at a conference workshop on Planet, more revision occurred. At each stage, the aim was to produce suitable ideas at the level of individual patterns, combined with a coherent arrangement of the various patterns. Overall, the process led to many patterns being culled, others being created, and all being revised.

5.3 Background: The Specialised Requirement of Software Internationalisation

5.3.1 The Increasing Internationalisation of Software

Internationalisation of software is rapidly becoming a critical issue for software manufacturers, due to expanding connectivity, economic globalisation, and the ever-increasing adoption of technology in all corners of the planet. Software developers can no longer produce a version for their local market and expect equal success overseas without substantial modification. Instead, systems must exhibit functional and non-functional features which are appropriate for end-users' cultures and conventions.

There are many reasons to develop systems for diverse populations. The benefits to software manufacturers are obvious: an opportunity to boost international market share and simultaneously reduce risk by diversifying their served market. From users' perspectives, the software closely matches their needs and generates synergies by letting people in distant locations interact via a common protocol. Developing countries can also benefit from technology which facilitates low-cost customisation.

The idea of a single "universal" version is naïve. To serve users' needs effectively, applications should be tailored for different cultures. At the same time, common components should be developed only once. This requires developers to distinguish between needs which are common to all users and those which are culture-specific. Typically, two stages are involved: localisation-enabling (or "internationalisation"), and localisation [138]. The first stage creates a framework within the core system to enable the subsequent localisation stage (e.g. creating a database to store language-specific strings). The internationalised framework is then instantiated in the localisation phase, when culture-specific data (e.g. translated text) is "plugged in".

How can developers ensure foreign users' needs are addressed? As with the general discussion of usability in Chapter 2, evaluation is not sufficient in this area. Developers must be aware of cultural factors upfront and design accordingly. Therefore, design patterns and other techniques would be useful. But in order to develop these techniques, it is important to understand those factors of culture which impact upon design.

5.3.2 Cultural Differences which Affect Software Requirements

The term *culture* generally denotes location and ethnic background. This is the definition used here, although it can be extended to include factors such as hobbies and lifestyle. People can therefore belong to more than one culture. Conversely, an individual nation can consist of numerous cultures. Since software requirements should be defined to help users achieve their tasks more easily and efficiently, it is necessary that requirements analysts spend time understanding the user. Culture is important in requirements analysis because it has a profound influence on the user's needs and characteristics. It is therefore useful to establish

a framework for modelling cultures.

A useful starting point is suggested by Yeo's [230] dichotomy of overt and covert factors [1]. Overt factors are tangible, obvious characteristics of a culture, such as calendars, units of measure, and character sets. In contrast, covert factors are vaguely-defined, complex aspects of a society which are easily misunderstood by outsiders and are sometimes so subtle that they may go unnoticed altogether. Examples include verbal and non-verbal communication style, the meanings of symbols, and problem-solving techniques. To document cultural factors in a way which supports development, it is desirable to extend this two-factor classification. In the remainder of this section, Yeo's classification is extended, with overt and covert factors refined into six and four sub-categories, respectively.

5.3.2.1 Overt Factors

In many ways, overt factors are somewhat arbitrary. Admittedly, some of these factors embody traces of cultural history; the fact that Australia shares the same calendar and date format as England provides a hint about its background. However, the information could be obtained directly anyway; overt factors *per se* tend not to offer deep cultural insights. They are the daily conventions under which a society operates and must be supported in software; most people do not particularly care how their currency is denoted, but once the unit of currency is set, then financial packages must cater for it. Since overt factors are based on convention, they tend to correspond to national boundaries.

Overt factors can be sub-divided into six categories, as shown in Table 5.1. Typically, they result in non-functional requirements. However, in situations where the user can customise software according to the factors, there will also be some functionality required. A functional requirement for a manufacturing system might state that the user can choose between metric and imperial units.

5.3.2.2 Covert Factors

Whereas overt factors reside on the "tip of the iceberg", there are also covert factors which hide below the surface. These factors may be very complex; even people from the culture itself may not be consciously aware of them.

As with overt factors, covert factors often translate into non-functional requirements. In addition, covert factors can have an impact on functionality, as discussed further on. Failure to identify covert factors early on can cause major problems upon release.

Covert factors fall into four major categories, as shown in Table 5.2. The first three categories resemble dimensions of culture discussed by Day [67] (cognitive decision-making style, knowledge representation, social control). However, the focus here is more on support for requirements engineers in contrast to Day's

[1] This thesis modifies Yeo's definitions to distinguish explicitly between cultural factors and consequential software features. Instead of treating an icon as a covert factor, it is viewed as a software feature whose interpretation will depend on certain covert factors.

Table 5.1. Summary of overt cultural factors for software internationalisation

Category	Factor	Example/Explanation
Time	Calendar	e.g. Australia: Gregorian; Japan: Emperor Era
	Timezone	Timezone varies according to geography
	Day Turnover	e.g. Islamic countries: Date changes at sunset [18]
	Work Cycles	People work at different times of day and on different days
Language	Translation	Words vary across languages, as does jargon across cultures
	Multilingualism	Some cultures have many people competent in more than one language
	Grammar	e.g. English: Subject-Verb-Object; Japanese: Subject-Object-Verb
Writing	Character Set	e.g. English: Latin alphabet; China: Pictorial alphabet
	Collation/Sorting	Character comparisons depend on alphabet
	Hyphenation	e.g. German: hyphenation changes spelling ("kuckuck"→"kuk-kuk" [215])
	Text Direction	e.g. English: Left-to-right; Hebrew: Right-to-left
	Special Characters	e.g. English: 1^{st}, 2^{nd}; Spanish: 1^{o}, 2^{o} [221]
	Case	e.g. Parisian-French: é→É; Canadian-French: é→E [221]
	Regular Expressions	e.g. "[a-zA-Z]" does not match all letters in most alphabets [138]
Measures	Currency	e.g. Portugal: 1$23; Sweden: Kr 12:- [78]
	Physical Measures	e.g. U.S.: Inches; Asia: Centimetres [221]
Formatting	Numbers	e.g. Belgium: 1.234,56; Japan: 1,23456 [78]
	Date/Time	e.g. U.K.: 31.1.95; Sweden: 95-1-31 [78]
	Address, Telephone No.	e.g. Austria: 1234 56 78 90; Italy: (12)3456789 [78]
	Number-Rounding	e.g. U.S.: 123.456→123.46; Argentina: 123.456→123.455 (digit not always dropped) [221]
External Systems	Standard Page Size	e.g. U.S.: 8.5" x11"; Europe: 8"x13" [215]
	Platform	Locale-specific platforms affect assumptions about resource consumption and service provision

Table 5.2. Summary of covert cultural factors for software internationalisation

Factor	Brief Explanation
Mental Disposition	How people perform their work, solve problems, make decisions. For example, affects suitability of direct-manipulation UI.
Perception	How people interpret and make sense of stimuli. For example, what types of icons are appropriate to represent certain concepts.
Social Interaction	Conventions of communication within a culture. Affects design of multiple-user systems and can also impact on human-computer dialogue.
Context of Use	Environmental context in which user performs tasks.

attempts to study cultural bases of interface acceptance. This perhaps suggests why Day's dimensions are more specific, and is the reason we include an additional factor–context of use–which is more of a pragmatic consideration than the other three.

5.3.2.2.1 Mental disposition Theories of human-computer interaction which are grounded in Western psychologies do not necessarily generalise to other cultures. Differences in mental disposition bring into question concepts of usability which are sometimes elevated to the status of "laws". Although many usability specialists advocate an `Undo` function to allow exploration, Ito and Nakakoji [116] claim that Japanese people regard trial-and-error as "tedious and time-consuming". This does not eliminate the need for reversible functionality in Japanese software, but it does suggest a lower priority. Furthermore, a finding such as this might suggest that facilities such as multiple-undo or redo features should be abandoned altogether if they call for significant development effort.

Consider direct manipulation, a veritable standard in many modern systems. A large motivation is the assumption that the user, rather than the computer, should be in control. However, it may be that some cultures are more accustomed to reacting instead of initiating, and would perhaps prefer a menu-driven system. Others may show a preference for command-line interfaces which provide even more control than the commonplace GUI.

A study by Evers and Day [77] demonstrates the complicated interaction between culture and mental disposition. They found that Indonesian and Chinese users differed in their design preferences towards a hypothetical computer program; the former group judged the software predominately on the basis of usability, while the latter group was persuaded more by usefulness. The finding suggests that, as well as designing culture-specific features, developers must even be aware of which aspects will be most important to each culture.

5.3.2.2.2 Perception Variation in life experience means that the same stimulus can lead to vastly different interpretations. One important implication for HCI is that metaphors can be culturally dependent; metaphors are supposed to relate concepts back to everyday experience, but everyday experience varies widely among cultures. The well-known Macintosh trashcan caused confusion to some Britons, who mistook it for a mailbox [215]. This is an example of users incorrectly identifying an object. Sometimes, an object will be correctly identified, but there will be differences in the concepts it connotes. Red represents danger in many Western societies but happiness in China [200]. Chinese users of Western software are just as likely as American users to correctly identify the colour of a red icon, but that icon might convey a very different message.

5.3.2.2.3 Social interaction Because software involves interaction between human and computer agents, it follows that social rules of the user's culture should be considered. Some cultures prefer to be more precise during communication than other cultures, for instance, and this may affect how much information a user is willing to enter to perform some task. Non-verbal communication, such as body language and facial expression, might also be considered. These could affect how a user inputs data (e.g. data gloves) and how the system presents itself to the user (e.g. hand gesture icon).

Social interaction also comes into play when the broader social setting of software usage is considered. Belge notes, for instance, "it is impolite to 'beep' in Japan, since this calls attention to a possible error on the part of the user" (p.23) [18]. The situation becomes more complex when the system must support numerous users interacting together, because then it must facilitate the forms of interaction appropriate to the culture, e.g. meeting styles can vary according to culture. Some cultures place greater value on authority, hence meetings are more likely to be led by a superior figure, with others mostly acting in a reactive manner. This type of variation should be considered when designing the functionality, user-interface, and dialogue among human and computer agents.

5.3.2.2.4 Context of use The environmental context in which a user operates can vary across locations, and therefore impose a range of constraints. Automatic Teller Machines in Sweden, for example, are built with large buttons to help people wearing thick gloves [173]. Similarly, building architectures and organisational hierarchies will affect requirements for facilities such as reporting and access rights. While Americans typically have their own workspace, Japanese offices are more open [78]. Workers in open offices probably have less need for e-mail, but are more liable to embarrassment from audio warnings after errors.

5.3.3 How Cultural Factors Impact on Requirements

Developing culturally-aware requirements specifications means more than just declaring who will use the system. As with other user characteristics, this information must be reflected throughout the requirements. Overt factors generally map isomorphically into non-functional requirements. For example, the knowledge that users require measurements in metric or imperial format simply means that the software must support both of these formats. The only other issue is how users can change their preferred measurement method. However, this becomes more of an issue about customisation than internationalisation itself.

The situation is radically different for covert factors. Obviously, these factors can affect surface-level features of the user-interface, such as images, colour schemes, and auditory cues. However, more profound effects also arise. Nielsen [171] reported on a French educational product which enables users to annotate poems. In some countries, it would be appropriate to allow children as well as the teacher to annotate, but in France, this capability was limited only to the teacher. This example illustrates how cultural values pertaining to authority and education can influence a central functional requirement.

Likewise, covert factors can affect non-functional requirements. Consider how one type of non-functional requirement — performance — depends on who will be using the system. First, a notion of suitable performance depends on how users achieve their task, i.e. their mental disposition. Some people might prefer to spend most of their time planning potential actions without directly interacting with the system. For such users, performance requirements might be less overall, but higher in the worst case due to occasional burst periods. Second, a notion of acceptable time to perform a task depends on how cultures perceive time and how long they are willing to wait. Third, the context of usage must also be considered. An employer will be more tolerant of slow software if operators are being paid salaries which are small compared with the cost of development, or if typical work processes in that culture do not place severe demands on performance.

5.3.4 Implications of Software Internationalisation

The research into internationalisation described above led to several recommendations for practitioners and researchers. Not all relate directly to design reuse, but the points have nevertheless been included as they illustrate the overall context in which this thesis fits. This section describes implications for practitioners

as well as for researchers. Most recommendations to practitioners are also reflected in the advice provided by the pattern language described later on.

5.3.4.1 Implications of Internationalisation for Practitioners

- **Anticipate future markets early on**

  The localisation-enabling process suggests that in many cases a large amount of effort can be saved by catering for cultures in advance. If the current version will not be shipped overseas, but a future version might, then planning can save substantial effort. Providing for an unanticipated culture may necessitate changes to code which is distributed across the entire program. For example, consider what happens if text format changes from ASCII to Unicode. In ASCII, a character comparison might simply use less-than and greater-than operators. Each comparison would need to be replaced by a function to accommodate Unicode. Had Unicode been adopted from the start, the upgrade would be minimal.

  Issues: It is difficult to predict who will use our software in the future. One solution would be to localise for every culture in the world. However, it is expensive to engage in broad localisation and developers should be aware that a little forethought goes a long way to reducing the cost. The potential cost of adding conditional structures at every point where the program requires some text is high compared with the cost of simply reading text from a database.

- **Recruit local expertise**

  There is no substitute for growing up and living daily in a culture. Thus, local experts are bound to be an excellent asset for specifying international systems. At the very least, they should be capable of reviewing a product and spotting obvious errors.

  Issues: It is difficult enough to find professionals competent in requirements and usability engineering. The additional requirement for local domain expertise increases the challenge. Smaller companies who do not have the luxury of subsidiaries and foreign partners may find this impractical.

- **Prototype and test with target markets**

  It is risky to perform usability testing on one user group and then expect general application of the results. Each culture-specific version should really be the subject of an entirely new testing process if the results are to be meaningful [171].

  Issues: Usability testing is expensive and testing for each version multiplies the cost. Effective utilisation of the internet may help here. A more overwhelming problem is the validity of usability tests. Studies in some cultures have found low correlations between subjective and objective assessments (e.g. [136]). At the very least, this suggests the importance of objective measures. It also brings into question the generalisability of Western-style tests and serious testing efforts therefore require the assistance of local experts.

- **Accumulate information acquired about cultures in an organisation-wide repository**

  Information which has been learned about a culture through development and testing should form part of an organisation's memory. Expensive resources are wasted if this information is not harnessed for other projects and future versions. Texts such as Fernandes [78] can provide a good head-start for information.

  Issues: It is impossible to be completely accurate, and publicly-available information is lacking. The key for an organisation is to record enough detail as to be genuinely useful for specifying requirements.

- **Analyse the trade-off between effort and quality of localisation**

  The extent to which you localise depends on the value you place on meeting the needs of targeted users. In some circumstances, it may be appropriate to adapt the users to the system, by selection or training methods, rather than vice-versa.

  Issues: Cultures vary not only in their preferences, but also in the value systems on which their preferences are based. Evers and Day's experiment [77] showed that cultures differed in relative preference for the variables of usability and usefulness. The complication is that usability and usefulness are themselves affected by culture. Additional effort will improve usability for a given culture, but some cultures care less about usability than others. The trade-off is complex because the underlying variables (and not just their values) are culture-dependent.

5.3.4.2 Outstanding Issues in Research

- **Developing a culture knowledge base**

  Software processes and tools for dealing with internationalisation are useful, but the major aid to developers is the knowledge base; that is, an understanding of the ways in which particular cultures vary. To complement individual practitioners gathering this information, researchers can contribute public, empirically-obtained, knowledge. Empirical studies with this goal in mind (e.g. [77], [28]) have been limited to date. This thesis does not contribute new knowledge about cultures *per se*, but aims to facilitate the usage of such knowledge (next point).

- **Accessing the knowledge base**

  Exactly how we present and contribute information to our knowledge base about cultures is also a burgeoning issue. Traditional printed style guides, with descriptions of each factor for each culture, are static, one-way media and unsuitable for widespread development. Instead, information should be stored in ways which facilitate dynamic updating from a variety of sources. The Planet pattern language (Section 5.4) describes a shared knowledge base, and Chapter 6 expands on this idea. The basic notion is that developers and users from around the world can access a web-based database and also provide information for it.

- **Engineering international software**

  Most work on producing globally-useful software was initially centered on implementation issues. Researchers have begun to discuss the implications of internationalisation for human-computer interaction. Over the next decade, we will need to devote more attention to the software-engineering aspects of internationalisation. As one example, consider the complexity of including internationalisation in the software development lifecycle [191]. The localisation-enabling phase must be integrated into the regular development phases. Then, the localisation itself may be carried out in parallel at locations scattered around the world, by engineers from different cultures, for users from different cultures. Aside from lifecycle management, other considerations include apportioning of requirements, quality assurance and verification mechanisms. The Planet language, with its inclusion of process issues, is a step in this direction.

- **Facilitating Computer-Supported Cooperative Work (CSCW)**

  CSCW systems are becoming increasingly important. Social interaction can affect single-user systems and CSCW systems for users of the same culture, as discussed earlier. Further research into cross-cultural CSCW is also warranted. Technical and ergonomic complications above and beyond the basic internationalisation problems are introduced. The various tailored versions must somehow work together in a cohesive manner. Cross-cultural CSCW is beyond the scope of this thesis.

- **Amortising cost for users with other specialised needs**

  "Accessibility" refers not only to international users but also to users with other specialised needs, in particular disabled users. There is an overlap here, because the user-base of assistive technologies is relatively small in any single culture, especially when one considers that the nature of disabilities varies widely. It may be cost-effective to produce core solutions which address the needs of a particular disability, and then customise it according to culture. This would effectively distribute the cost of supporting disabled users across different countries. This area is not directly explored in this thesis, although any work aimed at reducing internationalisation effort would be likely to improve the situation.

5.4 The Planet Pattern Language: High-Level Patterns

5.4.1 Introduction

Software internationalisation is about making software accessible, usable, and useful to people from different cultures and environments. It encompasses all of the cultural factors discussed in Section 5.3. As such, it address functionality, tasks, and user-interface design. The Planet pattern language provides a high-level view of the concepts and processes which can improve design quality of internationalised software. It is a usability-oriented pattern language which nonetheless takes into account software engineering issues. The

high-level patterns relate to the organisational process issues of modelling cultures, as well as user-interface design and functionality. Low-level patterns, which help designers effectively implement these high-level patterns, are deferred to an extended version of Planet (Chapter 11).

5.4.2 General Principles

Section 5.1 noted that the patterns in a language are related to each other not only via delegation relationships, but also by sharing a common set of underlying principles. It is useful to explicitly state the principles on which a pattern language is built, and in Planet's case there are five guiding principles. The principles arose from the analysis of software internationalisation discussed in Section 5.3 and are explained below.

5.4.2.1 Designers must Acknowledge Cultural Diversity

The most overwhelming implication of the identified cultural factors is that they affect what is meant by suitable design. As discussed in Section 5.3, cultures vary on many dimensions. Some overt dimensions, such as language issues and currency, are vital. Others, such as date formats, may have less personal significance, but still improve familiarity and avoid confusion. Covert factors may be more difficult to implement, but designers ignore these factors at their own peril. These factors are intimately associated with the thought processes and social workings of users, elements which are fundamental to user-centered design.

5.4.2.2 A Universal Version is Unrealistic

The language barrier is one obvious shortcoming for a universal version, and there are actually more subtle problems as well. Principles of human-computer interaction (HCI) are largely based on Western psychology, and may not fare well when covert factors of other cultures are considered. Even if well-established principles are relevant to another culture, some work is still required in order to apply them. One example has already been discussed in Section 5.3: the guideline of using metaphors, present in many popular style guides (e.g. [7]). If metaphors are based on everyday experience, and everyday experience varies across cultures, then metaphors must also vary across culture.

Utility — the net benefits derived from using the product — will also suffer if only one version is produced. Software utility depends to a large degree on its interaction with the domain. Just as everyday experience varies between cultures, so too does a given domain. Business rules, such as taxation legislation and currency conversion, may lead to new functionality. Likewise, differences in teaching philosophies dictate the acceptability of an educational package [171].

5.4.2.3 Every Person has Individual Needs

Labelling users according to culture can lead to stereotyping [28], thus denying them the chance to use software fitting their idiosyncratic preferences. In general, a culture-specific solution will be closer to a user's idiosyncratic preferences than a generic solution. We can improve the situation further by allowing the user to express their specific preferences, and supporting interfaces which may contain elements from more than one culture. The software medium is well-placed to exercise this level of flexibility, as opposed to physical devices. A traditional book written in French cannot suddenly transform itself to a Japanese text. Planet attempts to ensure that internationalisation is performed in a flexible manner.

5.4.2.4 Enable Then Localise

A development principle (mentioned in Section 5.3) is that internationalised software should be developed in two broad phases. In the *internationalisation* phase (also called *localisation-enabling*), databases and other structures are set up within the core system. These structures are then populated in the *localisation* phase, when translators, graphic artists, and others decide what is appropriate for a particular user community. This principle helps to focus development staff on their areas of expertise; technical staff work on internationalisation and localisation-enabling, while experts on particular cultures deal with localisation. Of course, aiming for a complete separation is idealistic, but the aim should be to separate these stages as much as possible.

5.4.2.5 Software Developers should Reuse Knowledge about Users

As was mentioned in Section 5.3.4.1, developers can save valuable resources by capturing knowledge about cultures. A principle of Planet is that it is useful to systematically construct and maintain an organisational memory relating to different cultures and appropriate design information for those cultures.

5.4.3 Pattern Language Overview

Planet consists of thirteen high-level patterns divided into four categories:

Culture Modelling Patterns are concerned with the ways developers can characterise and document information about cultures. A software project requires that an `Export Schedule` is identified. The schedule will reference a number of `Culture Models`. These `Culture Models` summarise knowledge about a culture, and can apply across the organisation. The format of the `Culture Models` is a `Vector Metamodel`. The culture models are aggregated together in an `Online Repository`.

System Patterns consists of a sole pattern, `Multicultural System`, which guides the mapping between cultures and features within a particular application.

System Feature Patterns are the general features present in a internationalised program. `Flexible Function`, `Elastic User-Interface`, and `Targeted Element` deliver functionality, user-interface structure, and user-interface features which are targeted towards the cultures mentioned in the `Export Schedule`. Since specialisation is not always feasible, `Universal Default` reminds developers to provide defaults suitable for a wide audience.

Feature Configuration Patterns describe design concepts which enable configuration according to culture, but in a way which accommodates users' idiosyncratic preferences. `Cultural Profile` suggests the system store profiles of values for each supported culture. `Citizen ID` describes the mechanism by which the user's culture is identified. `Integrated Preferences` addresses the presentation of culture-related preferences to the user.

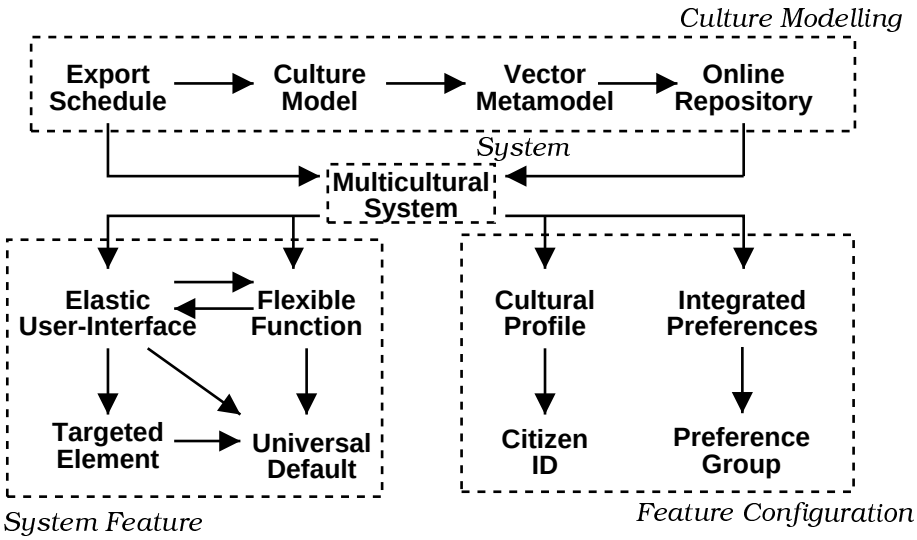


Figure 5.1. A map of the Planet pattern language. Arrows indicate process flow, i.e. the Resulting Context of the source pattern suggests the destination pattern may be used to solve a problem arising from the application of the source pattern. The boxes identify four pattern groups: culture modelling patterns, system patterns, system feature patterns, and feature configuration patterns.

Figure 5.1 provides an overview of the patterns. Note that the cycle between `Flexible Function` and `Elastic User-Interface` reflects the idea that functionality and user-interface should be iterated until satisfactory. The language begins being used when a developer is presented with a project that delivers to multiple cultures, or may do so in the future. For a brand new project, `Export Schedule` is an appropriate starting point. If there are no internationalisation projects, but some are expected in the future, developers may begin capturing relevant knowledge with `Culture Model`.

5.4.4 Descriptions of High-Level Patterns in Planet

This section lists all of the high-level patterns in Planet.

Pattern 5-1. Export Schedule

Context: You are beginning a project where users will come from a wide range of cultural backgrounds.

Problem: Internationalisation is a resource-intensive exercise.

How can you allocate resources at a level befitting the needs of each culture, and at the same time ensure that no more attention than necessary is devoted to any particular culture?

Forces:
- You should know which cultures will use your software, otherwise you cannot provide features which meet their needs.
- It is often too expensive to produce for all audiences at the same time. Sometimes, a local version is prepared first and overseas users are considered later on, if at all.
- Introducing a new culture may not be a simple localisation procedure. The new culture may necessitate changes to the core product and this can be painful. As one example, different cultures use different rules for sorting words. An English-language implementation can sort using the "<" and ">" operators, but what if a culture with a different alphabet order is introduced? If the comparisons are scattered across the code, then a programmer will have the unenviable task of replacing each one with a generic comparison function.

Solution: Produce a schedule which indicates *when* each target culture will be supported, and *how important* it is to support each target culture. Some cultures will not use the system until version 3.0 is released, but these cultures should still be considered when version 1.0 is developed. The `Online Repository` can be consulted to determine which factors vary across the supported cultures. The localisation phase can then address this variation.

In most businesses, interaction with the marketing department will be essential. When considering attractiveness of foreign markets, consider size and value of customer demand, competitive importance of the market, and availability of expertise [191].

At this stage, establish the importance each culture places on different aspects of software. For instance, an empirical study by Evers and Day [77] found that Indonesian subjects, compared with Chinese subjects, were more attracted towards usability than certain other qualities[2]. Since you cannot maximise every quality, this kind of information will tell you which areas to focus on for a

[2] It must be acknowledged that such studies represent a generalisation, since, in this case, there is much diversity *within* the Chinese and Indonesian cultures. The study nevertheless supports the argument that perceived importance of software attributes is culture-dependent.

given culture. Empirical studies can obviously not be conducted for each culture, but there are certain cues which will enable a rough guess. To understand the importance a culture places on different software features:

- Consult stakeholders from each culture.
- Consider whether certain products have succeeded in their target markets due to a particular feature which their competitors lacked.
- Determine if change requests from a particular culture exhibit recurring themes.

Rationale: Identification of cultures in advance is a prerequisite for a functional core and internationalisation framework which are flexible enough to make changes relatively straightforward.

Examples: Searching for target markets, choosing how intensive the operation will be, and planning future expansion is a well-established international marketing practice [120].

Uren et al. [221] suggest that marketing staff should develop a list of target countries, even if translation is not immediately required. Luong et al. [138] suggest that developers need to decide between a full localisation and a partial localisation, and also consider whether to ship overseas simultaneously with the local release.

Resulting Context: If your organisation has not already done so, start creating `Culture Models`. You may need to update existing `Culture Models` to reflect any new cultures being considered or to account for aspects of the cultures which are relevant to the new project. Once you have collected `Culture Models` into an `Online Repository`, you will be able to create a `Multicultural System` suitable for cultures mentioned in the schedule.

Note: For the remainder of this paper, the cultures mentioned in the `Export Schedule` are referred to as *Target Cultures*.

Pattern 5-2. Culture Model

Context: You are working on software projects with `Export Schedules` identifying which cultures will use the system.

Problem: The notions of usability and utility are culture-dependent. You have to understand the user before you can begin working towards these qualities.

How do you support development decisions which depend on information about specific cultures?

Forces:
- Extensive information may be required to support development decisions, necessitating potentially expensive research activity. This effort will have to be repeated on new projects involving the same cultures unless explicit attempts are made to reuse it.
- It is difficult — sometimes impossible — to reverse-engineer details about cultures just by looking at the resultant product. The Walkman has been successful in America largely due to the fact that people do not want to be disturbed by the outside world. but it was originally conceived by Sony's co-founder to help listeners avoid disturbing others [169]. An American designer who simply observed a Walkman would probably not be able to predict its designers' initial vision.
- Even if you can access process documentation such as meeting minutes, information about cultures will be scattered. This will be difficult to find specific information when it is needed during internationalisation and localisation.

Solution: Construct models of cultures which are relevant to your projects. When you discover new information about a culture, add it to the culture model. To improve opportunities for reuse, these models can, and should, be held in common by the entire organisation, rather than pertaining to a specific project. They are an asset which is refined over time, just like a software library or an estimation technique.

The culture model can also include issues regarding the development process. Luong et al. [138] discuss the English-language ability of the quality assurance engineers they deal with in Japan. Another culture-dependent process is user testing; some cultures appear to be more reluctant than others to criticise a product [105].

Rationale: Consistent culture models provide a central point for culture-specific details and also help developers compare and contrast cultures. If a developer notices that all of the cultures for a particular program use Latin characters, then the internationalisation phase need not consider the possibility of non-Latin character sets. Localisation will also be more efficient if similarities between cultures can be exploited.

Examples: There are several well-known culture models in anthropological and marketing literature. In "Understanding Cultural Differences" [101], Hall and Hall explain several culture-dependent variables, e.g. the level of detail people desire when presented with information, attitudes to personal space, whether or not people like to perform tasks in parallel.

Luong et al. [138] provide details on localising for Asia. Numerous business texts describe the intricacies of doing business with, or creating products for, a particular country [114] or region [66].

Resulting Context: Now that you've decided to create a culture model, you'll need a way to structure it. A culture model can be conceptualised in different ways; an extremely simple form is just an

arbitrarily-ordered collection of relevant documentation gathered from various projects. This pattern language focuses on one means of characterising a culture: the `Vector Metamodel` form.

Pattern 5-3. Vector Metamodel

Context: You have decided to create some `Culture Models`.

Problem: Culture models can be represented in different ways. The form you decide upon will determine who is capable of using the culture models, as well as the range of activities the models will support.

How do you represent all relevant information about each culture in a form which supports internationalisation processes?

Forces:
- A culture model should accurately reflect aspects of a culture which may impact on software requirements. It should facilitate the two internationalisation phases — internationalisation and localisation.
- To ensure that it is always up-to-date, a culture model should be easy for people to contribute to.
- Culture models need to be expressed so as to be easily-accessed and understood by people in diverse roles. This includes software engineers, human factors specialists, anthropologists, marketing staff, and users.
- It should be possible to compare and contrast culture models. A user of the repository should be able to quickly determine how cultures differ from each other, and also notice commonalities.

Solution: Determine the dimensions of cultures that interest you, and characterise each `Culture Model` as a vector with a value for each dimension (or "factor"). A culture model is then a description with each dimension defined for that culture, e.g. if you decide the dimensions are language, timezone, and date format, then *every* culture model in the repository will have those dimensions defined (definitions can be left blank if it's obvious or not yet known).

There are many potential sources for culture models. These include literature reviews, comments from local branches or consultants, results from requirements elicitation and user interviews, and observations of the system in action.

If desired, divide the dimensions into categories. For instance, Mahemoff and Johnston [146], following a suggestion by Yeo [230], discuss two broad categories — overt and covert factors. The former are obvious, tangible factors (e.g. date format), whereas the latter are ill-defined and potentially controversial. Overt factors can further be divided into several categories: time, language, writing, measures, formatting, and external systems (Table 5.1). Covert factors are composed of mental approach, perception, social rules, and usage context (Table 5.2). The overt factors could

provide a valid starting point for your metamodel. The covert factors are more abstract and in need of refining before they are placed into a repository. For example, the Perception factor indicates icons should be chosen according to cultural understanding. Instead of including a broad "Perception" factor, it would be worthwhile storing a field for each major icon type you are likely to need. For instance, a "Mailbox Appearance" field if your software provides email. With all of these factors, you will need to filter them according to your organisation's needs.

Rationale: The simplicity of this approach makes it accessible to people working in various roles. The metamodel also has a close mapping to software concepts and is closely related to the `Name-Value Pairs` pattern [196]. A culture vector can be viewed as a data structure, with dimensions and cultural information forming key-value pairs. In some instances, this could be mapped directly into a class or a database table. At the very least, it shows that the classification scheme is well-understood by developers.

If cultures are classified according to the same dimensions, then it is also easy to see where they are alike and where they vary.

Is it overly simplistic to characterise a culture as merely a set of key-value pairs? This depends on how the model is used. While the approach does not lead to a highly cohesive view of a culture, it is certainly a practical way to enable people to use and update a model. Also, Hoft [108] discusses a number of metamodels, all resembling this pattern; each lists several factors by which cultures vary. They mainly vary in the ways they classify the factors, and the amount of attention paid to interaction among the factors.

To avoid oversimplification, the value for each field can be a discussion of options, e.g. for units of distance in America, imperial units are used in some contexts and metric units in other contexts.

Examples: Del Galdo [69] presents several attributes of cultures. She divides them into categories, depending on how closely they relate to the user-interface. The factors include character sets, numeric formats, and currency formats (non-UI), as well as icons, colour, and screen text (UI). These factors could form an embryo for a metamodel by which to document a culture model. Hoft [108] discusses several well-known attempts to characterise a culture. Generally, these models are based on a number of dimensions, e.g. collectivism versus individualism, attitudes towards time.

POSIX has a standard way to define a locale, according to ordering of characters, output formats, message strings, etc [232]. Java's Locale class and other libraries provide ways for a developer to identify a resource (*WelcomeIcon*), and then show how it relates to each supported locale (*Hello.jpg*, *Bonjour.jpg*). There is a similar framework available for C++ [131].

Resulting Context: : You have a consistent structure for your culture models. As you gather culture models according to the vector metamodel, you'll probably want a way to organise them together.

To do this, establish a `Online Repository` based on a consistent `Vector Metamodel`.

Pattern 5-4. Online Repository

Context: You have begun to maintain `Culture Models` according to the same `Vector Metamodel` (i.e. same factors for each culture).

Problem: As you start accumulating `Culture Models`, you will realise the need to organise them together.

How can a collection of culture models be organised in a manner which is useful for software projects?

Forces:
- Organisation-wide Culture Models avoid duplication; it is feasible and desirable to transfer information learnt from one project into other projects.
- Information about cultures is often discovered in physically distant locations.
- If developers can't access models quickly and easily, the information will be ignored.
- If developers can't update models easily, the repository will lose accuracy over time.
- Cultures have relationships with one another. We should be able to capture associations, such as one group being a sub-culture of another group.
- It is useful to look up a specific `Culture Model`, but there are many other ways people might like to access information during the internationalisation process. Depending on the task at hand, developers may wish to explore the information in unanticipated ways, e.g. comparing two cultures, considering a single factor across numerous cultures.

Solution: Create an online repository, accessible to the entire organisation. Compose it of `Culture Models` all based on the same `Vector Metamodel`.

The following guidelines make it easy to *access* information in the repository:

- Provide browsing facilities which present each culture and factor. Let the user select a factor (and show how each culture varies on it), a culture (and show all of its factors), or a combination of both.
- Provide facilities to search the `Culture Models`.
- Link from one model to another if it helps to demonstrate a point of similarity or difference.
- Link to the original artifacts if they are online, or identify sources if they are not.

The following guidelines make it easy to *update* the repository:

- Facilitate discussion among contributors, e.g. via a mailing list or on the repository system itself.
- Make one individual responsible for managing the overall repository. This role implies promoting the repository within the organisation and chasing results when it is known that valuable information has been attained but not entered.
- For each `Culture Model`, make one person responsible for maintaining it. Some `Culture Models` may be maintained by the repository manager, but an expert on the culture is a better candidate.

Large repositories can group `Culture Models` together, e.g. according to continent. This approach can follow the `Composite` pattern [88], e.g. a model of Europe contains its own information and also contains child models (France, Italy, etc.). When a user looks up a model, information about its ancestor models are also shown.

Rationale: Like pattern languages, repositories of this nature help people reuse existing, tested, knowledge. Reuse of proven concepts involving human-computer interaction are particularly helpful, because people's reactions can be difficult to predict. In the case of interaction with international systems, the case for reuse is even stronger, because more work is required to obtain original knowledge (e.g. travelling overseas, establishing partnerships with foreign consultants).

The concept of grouping cultures serves a couple of purposes. Firstly, it emphasises commonalities among similar cultures, more so than if both cultures had separate, identical, entries. Secondly, there is obviously a great practical benefit: it prevents the need to create and maintain duplicated entries.

Examples: Fernandes [78] contains some tables showing factors versus culture. However, the text stops short of exhaustively listing this information. One table might show cultures A, B, and C, and a table on other factors might show A, D, and E.

Ito and Nakakoji have prototyped a system for retrieving culture-specific details [116].

The authors are currently undertaking a project to build a web-based repository. The intention is that developers and users from around the world will use and contribute to the database.

Resulting Context: The repository enables developers to easily access a corpus of culture-specific information. You can use this information to specify a `Multicultural System`.

Pattern 5-5. Multicultural System

Context: You have created an `Export Schedule` for a new product and it shows that a diverse cultural base must be catered for.

Problem: How can you create software which is flexible enough to meet the needs of culturally-diverse users?

Forces:
- Your system needs to be general enough to accommodate the users' needs and characteristics, which depend somewhat on culture.
- System features can take on one of several forms. For example, a text field may appear in English or French. For most features, it is not realistic to provide a single form which will satisfy every culture.
- The user is mainly interested in the form which relates most to their own culture, and it is therefore undesirable to provide more than one form simultaneously.
- A software development team is likely to impose its own cultural biases on the target system, best intentions notwithstanding.
- Users from different cultures may use the same software. Your software will become inaccessible to some users if you permanently fix each feature.
- Software should not stereotype the user as belonging to one particular culture. It should let the user tune settings to meet their own particular needs.

Solution: Produce a single version of the software with each all instances of each culture-dependent feature present. Let the user set their preference for each feature.

The user can then mix cultures by choosing forms of features which have been created with different cultures in mind, e.g. an American can leave text in English, but select a European paper format.

Sometimes, it is impractical to provide all forms of all features. This may be due to space constraints, downloading times, or commercial considerations. In these cases, it is tempting to produce a separate version for each culture. However, a better solution is to keep the software flexible, so that users can add and remove the forms pertaining to a particular culture.

Examples: Most major operating systems and applications provide versions tailored to specific cultures. Typically, the main difference is language. Other differences include measurement units and currency. Often, it is possible to add culture-specific extensions to the base system, e.g. help files in foreign languages and international fonts.

Unicode, supported in HTML 4.0, is a text format containing characters from every major alphabet in the world. This allows information pertaining to various cultures to be manipulated by the same application.

Resulting Context: You will need to produce tailored features, specifically `Flexible Functions`, an `Elastic User-Interface`, and `Targeted Elements`. As you develop these features, you will need to develop `Cultural Profiles` to specify which features relate to each supported culture.

`Integrated Preferences` will enable the user to specify both culture-specific and culture-neutral features.

Pattern 5-6. Flexible Function

Context: You are producing a `Multicultural System` and an `Online Repository` has been established. You have begun to specify the user-interface according to `Elastic User-Interface` or you feel that it is more appropriate to specify functionality before the user-interface.

Problem: A culture-sensitive user-interface may contribute to *usability*, but it is still possible that the software does not support the tasks users would like to perform, i.e. the functionality lacks utility. These tasks and the context in which they occur can be related to culture.

How do you ensure the software performs the functions that are meaningful and useful to people from different cultures?

Forces:
- Software is typically written with specific domains in mind, whether broad (e.g. a spreadsheet) or narrow (e.g. a code inspection tool).
- Domains — whether broad or narrow — are not homogeneous with respect to culture.
- Usability is also influenced by the user's culture, and usability derives from more than just the user-interface. Flexible searching, for instance, cannot be achieved just by applying `Elastic User-Interface`.

Solution: When you generate a new function, check if it is culture-specific, and if so, refine it to meet the needs of your target cultures.

Break the requirements phase into several smaller stages, where each stage consists of creating and/or refining a small set of related requirements. This way, you can progress incrementally, so that each stage can take into consideration the cultures mentioned in the `Export Schedule`. At each stage, consider the impact on the target cultures. Some cues which might suggest culture is an important factor include:

- a requirement depends upon legislation (a taxation rule).
- a requirement implies an organisational role (only certain people are authorised to shut down the assembly line).
- a requirement is based on a philosophical stance (a teacher can annotate text but a student cannot [171]).
- a requirement is underpinned by certain ethical values (an employee's email may be reviewed by supervisors).

This process may generate new questions about cultures which the repository should be consulted to solve. If it cannot solve them, seek the most important answers by alternative means (e.g. interviews with domain experts) and update the repository. Once the answers to these questions are known, you will be in a better position to refine the requirements. Your initial idea for a requirement may form the basis of a suitable default (a `Universal Default`), but you may need to extend it to satisfy all target cultures.

Examples: Time-keeping variations imply more than just differences in user-interfaces. Each supported calendar format requires functionality dedicated to handle standard operations (e.g. finding weekday from date, incrementing date). The situation becomes even more complex when differences in other areas, like timezones and work cycles, are considered.

Sometimes, a standard function is undesirable or not worth the effort. Japanese dates are stored according to the number of years of the present Emperor's reign. Lotus learned that a feature allowing users to return the year back to zero was offensive, because it implied the Emperor's mortality, and removed the feature [78]. Another Japanese example is the suggestion that Japanese users prefer planning ahead instead of trial-and-error [116]. There may be a case for scaling down the *Undo* function, or not implementing it at all, if a large proportion of users are Japanese.

Currency differences can lead to complicated functionality, especially when conversions are required. The introduction of the Euro is a familiar example. A more complicated example is the currency recorded for transactions. Some countries denominate imports in their own currencies, while other countries denominate imports in US dollars.

Nielsen discussed a French educational product which enables teachers to annotate poems [171]. He noted that in some countries, it would more appropriate to give students the same ability. The decision to include this kind of functionality rests on cultural values such as attitudes to learning and authority.

Resulting Context: Iterate between this pattern and `Elastic User-Interface` until you are satisfied with functionality and user-interface structure. Establish a `Universal Default` for each function in case the user's culture has not been specifically catered for.

Pattern 5-7. Elastic User-Interface

Context: You are producing a `Multicultural System` and an `Online Repository` has been established. You have a solid understanding of general functionality after applying `Flexible Function`, or you feel that it is more appropriate to specify the user-interface before concentrating on functionality.

Problem: The same functionality can be presented to the user in different ways.

How do you create a user-interface which can be used easily and effectively by all targeted cultures?

Forces:
- Different cultures use different schemes for representing information. Text direction, for example, can be left-to-right, top-to-bottom, right-to-left. Relative positioning of graphics and highlighting mechanisms also vary [207].
- Some cultures want more information, or different information, than others. Some cultures are more accustomed than others to inferring missing details [101]. Cultures may also vary in the nature of information required. The right feature cannot just be plugged in later on (e.g. a culture-specific icon or language-specific string).
- Human-human interaction varies considerably across cultures, and this can translate to style of human-computer dialogue, e.g. reactivity versus proactivity exhibited by participants.

Solution: Design the overall structure for the user-interface flexibly, so that UI elements can subsequently be re-defined and rearranged without massive design changes.

The precise details about what information you are presenting, and how it is arranged will not become clear until the localisation phase. Therefore, develop an elastic user-interface so that the user-interface is not fixed during the internationalisation phase.

As with `Flexible Function`, break the user-interface specification process into several stages, so that you can plan each subsequent stage in the light of the cultures being targeted.

Two general guidelines apply:

- Consult the repository or use other means (e.g. user testing) to discover how elastic the user-interface structure must be. A culture's writing direction can indicate the order in which material of a non-textual nature is also perceived. Also, text size will vary due to variations in alphabet, word lengths, grammar, and conventions. This will affect the space allocated for writing.
- Once the overall structure is present, you will have identified the UI elements. However, the precise details of UI elements depend on culture. Therefore, at this stage, identify which UI elements are culture-specific and leave them abstract, e.g. a button to delete a file could be called *delete-file* and its type simply declared as a button with a bitmap and a label. After this pattern has been applied, an artist performing localisation might create a cross for one culture and a skull-and-crossbones for another, with accompanying labels in the appropriate language.

Rationale: Complicated changes to the user-interface, such as reversing order of elements, can necessitate widespread code changes if they are not designed for. Also, a common problem is the introduction of

a language with text too large to fit in its usual location (e.g. a menu bar or dialogue box) [138]. This would be fairly trivial to prevent if it was known that the language was scheduled for introduction.

According to the Slinky meta-model [62], it is possible to split user-interface, human-computer dialogue, and functional core. Architectures based on Slinky should provide adequate support for developers who wish to create `Elastic User-Interfaces`. Coldewey's "User Interface Software" pattern language also covers separation of user-interface and domain [43].

Examples: Some programs enable text to be shown and inserted in right-to-left as well as left-to-right modes. Farsi support within the Vim text editor [225] lets users dynamically switch between orientations, and can simultaneously show one file in two different windows presented in separate orientations.

The initial website for some browsers depends on the region of the software.

The web supports various layouts. Horizontal orientation of layout (e.g. whether links are on left or right) is correlated with the orientation of text in the originating country [13]. It is worth noting, though, that an individual site generally does not provide more than one layout.

Resulting Context: Iterate between this pattern and `Flexible Function` until you are satisfied with functionality and user-interface structure. You can design specific features according to `Targeted Element`. Establish a `Universal Default` for the UI structure in case the user's culture has not been specifically catered for.

Pattern 5-8. Targeted Element

Context: You have prepared an `Elastic User-Interface`.

Problem: How can you create specific elements to plug into the user-interface framework?

Forces:

- The purpose, state, and workings of a user-interface element should be clear from its appearance.
- A user-interface element's appearance can be misinterpreted if the user's culture was not considered in designing it. An extreme case is when users are presented with foreign languages, but there are more subtle instances. The famous Macintosh trashcan looked like a postal box to Britons, causing great frustration when they tried to e-mail their work [215]!
- It is also possible for a feature to be correctly identified at a surface level, but still be completely inappropriate. Many applications use hand gestures to represent certain concepts, but some of these are offensive in some countries [78]. Similarly, a given colour can convey different moods, depending on the culture in question [200].

Solution: For each abstract element contained in the `Elastic User-Interface` specification, provide an instantiation targeted to each culture in the `Export Schedule`.

The `Online Repository` can be consulted for guidance. If it proves insufficient, investigate further and remember to feed results back in to the repository for future use.

Many elements will have a common instantiation for an entire group of cultures. This is where the idea of composite cultures discussed in the `Online Repository` can be useful. For instance, a group of cultures called "English-language" can share the same text (of course, this would be a compromise because it is preferable to have "USA-English", "Australian-English", etc.). Such a grouping system would need to be many-to-many rather than strictly hierarchical, i.e. a country such as England might belong to several groups, e.g. "English-Language", "Commonwealth", "Great Britain".

The `Online Repository` can help to identify features which are misleading or offensive. However, user testing is also essential since people from external cultures can overlook this kind of problem. In marketing, there are countless examples of brand-names and advertisements which were actually used, despite the fact that any lay-person could see the folly, such as products named after swear-words and products with unwanted functions (many examples are described by Meloan [163]).

Examples: This pattern is exemplified by any software that supports different languages, address formats, units of measurement, currency, etc. Tools and libraries like Java's MessageFormat class support the corresponding development process.

The culture-specific partners of the *Yahoo!* website are another example. They are organised in the same way as their parent, but contain information specialised to their culture, such as weather for certain cities and status of local stockmarkets.

Java's MessageFormat class allows programmers to specify a culture-specific format without committing to precise data. An English format can be "subject-verb-object", while a Japanese format can be "subject-object-verb". Then, only a single statement identifying the subject, object, and verb, is required. This avoids having two print statements whenever a message is generated.

Resulting Context: The system contains culture-specific elements, ready to be plugged in whenever a user of the appropriate culture requires them. Since it can be expensive to target every element to every culture, provide a `Universal Default` in case the user's culture has not been specifically catered for.

Pattern 5-9. Universal Default

Context: You have prepared functionality and the user-interface according to `Flexible Function`, `Elastic User-Interface`, and `Targeted Element`.

Problem: Localisation is costly and requires substantial knowledge about target cultures. Providing many features for many cultures can lead to work in the order of $m \times n$.

How do you optimise resources when there are many features (functions, user-interface arrangements, user-interface elements), each of which depends on a wide cross-cultural base?

Forces:
- Features should generally be tailored, because a single version of a feature is rarely adequate for all cultures. If the single version is targeted to one culture, other cultures may suffer; if the version is intended to be universal and not belong to any particular culture, it may be too general and suit no-one at all.
- Tailoring software features to a culture is an expensive, time-consuming activity. It may involve extensive research and requires local experts.
- It is not always feasible to tailor software for small cultures or to cultures that have unique needs that are difficult to implement.
- An `Export Schedule` cannot always correctly anticipate which cultures will use the software. People from unsupported cultures might move to targeted areas or download software from a website.

Solution: **For each culture-dependent feature in the target system, make a default which is universally meaningful.** The aim is to reach a fallback setting in case nothing has been tailored for some cultures. This will never be as good as a tailored feature, but it is better than producing a default which does not consider the cultural issue at all (e.g. one which is only suitable for the developer's culture).

The following guidelines will help when creating universal defaults:

- Provide text in the most familiar language among supported cultures.
- Endeavour to produce images which are easy to understand and free from cultural bias.
- Since not everyone will know the most familiar language, try to replace text, e.g. button labels, with meaningful images.
- The familiar language may still be non-native for many users. Avoid the use of jargon and concepts which would only be familiar to a subset of supported cultures.
- Avoid concepts which might offend some cultures.

For organisation-specific `Online Repositories`, it may be useful to include a "Universal" Culture Model in the `Online Repository`. This would contain organisation-wide defaults. Such a model would be less useful for a repository shared by the overall community.

Rationale: In some ways, this pattern resembles the naïve approach to internationalisation which claims that we should simply provide universal features. The difference here is that the pattern is seen as a fallback to help reduce development effort, and is certainly not claimed to be ideal from the user's point-of-view.

Examples: Numerous websites provide versions in two languages: one in the native language and one in English. English is being used as the language which will reach the most people.

Some icons and metaphors from popular software paradigms are well-understood in different cultures. Web browsers often use left and right navigation arrows. Word-processors and painting programs use a blank page to represent a new document and scissors for cutting. Help is often represented by a question-mark.

Resulting Context: There is now a default for each user-interface feature, to account for cases when the user's culture has not been specifically catered for.

Pattern 5-10. Cultural Profile

Context: You are designing a `Multicultural System`.

Problem: How do you handle configuration of features which are culture-specific?

Forces:
- A `Multicultural System` has a lot of choices because each feature has several culture-specific variants, all of which exist in a single version. Configuring can be time-consuming for users.
- Most users want to begin working on a product right away, rather than exert effort configuring it. There is a probably a good reason why the user has just downloaded or purchased the product, and loaded it up for the first time.
- Users may not be capable of specifying the appropriate settings for some variables, even in their own country. Imagine asking a user to choose among grammatical rules for a grammar-checker!

Solution: Provide a default profile for each target culture, a profile which specifies the value of each culture-dependent feature. As soon as the user specifies the culture, they can begin working right away, tweaking settings to their idiosyncratic preferences whenever desired.

An additional benefit is that users can dynamically switch between cultures. This may be of use when two people share the same application. It can also be useful to some users who work in different "culture modes". Some people think in one language while they work, but think in their own language during recreation. The phenomenon of "code-switching", i.e. alternating between languages, is common when someone has acquired technical skills in a foreign language [96]. Many

people in multinational firms may work with software in English, but perform personal functions such as online banking and email in their native languages.

Examples: The locale library in Linux defines several culture-related options, e.g. language, currency. However, instead of setting these individually, the user can simply set the LC_ALL variable, which ensures each setting is appropriate.

The Dell website [59] has different pages depending on country of origin. Various settings then relate to the country, e.g. support phone numbers, available purchase items.

Resulting Context: A number of cultural profiles are available. `Citizen ID` provides a mechanism for the system to determine which profile to adopt.

Pattern 5-11. Citizen ID

Context: You have an application with several `Cultural Profiles`.

Problem: Versions must sometimes be released separately for each culture, for technical or commercial purposes. More flexibility can be offered if details for all cultures can be incorporated into the same version. However, not all settings can be active at once.

How can the program determine the appropriate settings for a given user?

Forces:
- In the absence of other knowledge, the settings are predominately driven by the user's culture.
- The user's culture does not normally vary during the same session, or even across sessions.
- Users do not care about the distinction between the present session and future sessions. This is merely a technical reality because of machines being switched off, and limitations of memory and other resources. All a user needs to know is that they interact with the same program at different points in time.
- Information which is well-known to the user and unlikely to change should be represented minimally within the user-interface, if at all.

Solution: Determine the user's culture on first use and ensure that this preference persists into the future. Select the `Cultural Profile` based on the user's Citizen ID. Typically, the user selects their location from a menu of names or by clicking on a flag or map. The program may also obtain it from the operating system. Once the user has specified their preference, the choice does not have to be shown. It is sufficient to re-display the preference whenever the user starts the program, e.g. on the splash screen. This ensures they have not accidentally set the wrong culture or inherited someone else's preferences. A similar effect can be achieved with websites using persistent cookies or redirecting to a localised subsite.

When a user specifies or re-specifies their culture, this has the effect of re-setting culture-specific settings.

It may be possible to obtain the user's culture from the external environment — by querying the operating system or a related product. Use such data as a first guess and confirm with the user on first usage. For instance, if you present a menu of cultures, initially select the best guess.

There will need to be a mapping from each Citizen ID to a particular `Cultural Profile`. A one-to-one mapping is possible, but it will often not be practical. For example, Canada and USA might have the same `Cultural Profile`. The operating system might identify users by their country, but this will need to be mapped to the same `Cultural Profile`.

Examples: Modern operating systems usually ask the user about their location upon installation. They can use this information to set defaults for timezones, character sets, and other information. Other programs can subsequently query this data to set their own defaults.

Websites often let the user click on a flag or some text to set their language preferences. Alvarez et al. [5] designed a bilingual website which only contains links to the other language from each language's homepage. Since the user is not likely to change languages, this strategy optimises screen real estate.

Resulting Context: The users culture is known to the system and will be used to set initial values for culture-specific features, i.e. `Flexible Functions`, aspects of the `Elastic User-Interface`, and `Targeted Elements`.

Pattern 5-12. Integrated Preferences

Context: You have incorporated `Citizen ID` into the overall product.

Problem: The `Citizen ID` pattern will inform your programmer about the user's culture. It is tempting to pigeon-hole a user as a generic American, a generic Australian, etc., but users within the same culture actually vary widely.

How do you avoid stereotyping users?

Forces:
- A user thinks of herself as an American, an Australian, and so on; not as an "international user" [18]. This makes an "International Settings" dialogue a bad idea.
- A user's culture is more than just his ethnic background. A person's friends, hobbies, or work-place can also act as cultures. *Culture* is a fuzzy concept.
- There are many more reasons to provide flexible software aside from culture. The user's tasks and context are also vital considerations.

Solution: Integrate internationalisation-related preferences with general preferences. As discussed in `Citizen ID`, when the user sets their culture, this simply has the effect of setting internationalisation-related preferences to pre-defined defaults.

Examples: Two culture-related settings in word-processors are font choice and paper size. Rather than placing these in an "International" dialogue, the designers typically position them among semantically-related items. Font choice is usually near font style (Bold etc.). Paper size is usually near layout options (margin, headers, etc.).

Resulting Context: The program accounts for the user's culture. At the same time, it does not use their culture to force restrictions. You can help the user tailor their preferences by creating `Preference Groups`.

Pattern 5-13. Preference Group

Context: Your `Integrated Preferences` model identifies all features which vary by culture.

Problem: `Integrated Preferences` may contain many variables, culture-dependent and culture-neutral. **How do you structure the `Integrated Preferences` for presentation to the user?**

Forces:
- `Cultural Profile` improves efficiency by providing a quick mechanism to set a whole sequence of preferences in one hit. But the user must still tailor `Integrated Preferences` to their own needs.
- There are often many features to customise. Each word to be displayed is a separate culture-dependent variable. Therefore, customisation can be time-consuming for users.

Solution: Group related preferences, so that only one choice needs to be made by the user to set all of the preferences in the group to logically-related values. The typical example is a message group: the user sets the messages to "French", and all of the individual messages are set to "French".

In some respects, this is like `Cultural Profile`, which also maps one choice to a number of pre-defined preference values. However, this pattern relates to subsets of the preferences rather than the overall preference vector. Furthermore, the direct nature of the group is often culture-neutral. For instance, a setting of "I like Warm Colours" might lead to orange buttons, red menus, and yellow backgrounds. This is a choice relating to the user's favourite colour scheme rather than the user's culture. Of course, there is still an indirect relationship because certain cultures like certain colours. For this reason, it is useful for the `Cultural Profile` to contain pre-set `Preference Groups`. In this example, there will be a "ColourScheme" preference group object relating to warm colours, i.e. a ColourScheme object with the following settings: "Button=orange", "Menu=red", and "Background=yellow". An alternative ColourScheme object might be a cool colour scheme.

We can then decide on a default ColourScheme for each culture. For instance, we might reference the warm ColourScheme object as the ColourScheme in the Australian `Cultural Profile`. If we later decide Americans also like the warm colour scheme, we can refer to the same object in the American `Cultural Profile`. This is preferable to manually defining the colours for each widget type for two reasons: (a) there is less effort, since the colour scheme has already been defined; (b) the "ColourScheme" `Preference Group` contains semantic knowledge (i.e. American interfaces default to warm colours) which would not be conveyed if only the individual widget colours were defined.

Sometimes the user can reach inside the group and set individual variables. This might make sense in the colour example, e.g. the user might prefer a different button colour. This is still possible. Other times, the user would be unlikely to change variables inside a `Preference Group`. If the `Preference Group` represents translated text messages, the user would likely have no desire to tailor these.

Examples: Any application enabling the user to specify natural language exemplifies this pattern. The GNU locale environment, for example, lets the user set a LC_MESSAGES variable which causes a number of strings to be translated in a particular language.

The example application, Critique, in Chapter 11 contains an "Evaluation Style" preference. This determines how a score is calculated and what data the user sees; the user only sees data which affects the score. There are two views and two score algorithms, leading to four possible combinations. However, a design decision was made to couple these preferences via the group mechanism, so that only two of the four combinations are possible.

Resulting Context: The user can tailor their preferences faster. Developers can save effort by capitalising on the tendency for several cultures to prefer the same combination fo settings for some variables.

5.5 Discussion

Planet demonstrates the power of choosing the appropriate scope for a pattern language. A tight scope in one dimension can allow a broader scope in other dimensions. Planet is tightly-scoped in the dimension of specialised requirements, focusing squarely on the needs of users from different cultural backgrounds. Planet is also constrained moderately by target medium, with target systems generally being standard desktop applications or websites; the language would need refinement if it were applied to CSCW systems, for instance, since inter-cultural CSCW entails a whole new set of problems [45].

By making these compromises, it is apparent that Planet is well-placed to comment on a *broader* range of abstraction levels. The language enables a smooth integration between organisational process, supporting tools, and high-level software design. The patterns show how these entities interweave. For

instance, consider the advice of `Cultural Profile`: a developer should construct a profile for each target culture. The target cultures are drawn from the `Export Schedule`, a list of current and future user groups which is an organisational decision. Information about the software design and features must then be combined with information about the culture to design the appropriate profile. The profile itself is an entity in the software, meaning that `Cultural Profile`is a pattern of software design while `Export Schedule` is more a pattern of an organisational artifact.

The ability to transcend across levels of abstraction is one of the greatest benefits patterns have to offer. Certainly, Alexander's patterns ranged from global planning concerns down to detailed building construction (Section 3.2.1). By providing a coherent set of patterns at varying levels of abstraction, a pattern language can facilitate interdisciplinary communication within the organisation. The `Export Schedule` pattern can have input from marketing specialists and experts on particular cultures. When it comes time to produce a `Cultural Profile`, there can be input from software designers, usability specialists, and culture experts. The pattern language provides a common vocabulary for issues encountered during software development. More importantly, it paves the way for an interdisciplinary development process. It begins with a set of guiding principles and each of the patterns work together to show how implementation can adhere to these principles. The language does not just represent some snapshots of internationalised software; it is a set of concepts which work with each other to push development in a certain direction. Entities like `Culture Model`and `Online Repository` are central features of the language, in a similar manner to the design metaphors of `Tools` and `Materials` in Riehle and Züllighoven's Tools and Materials language [194].

The language covers organisational process, functionality, and conceptual design, but does not cover detailed software design. To provide further value to the interdisciplinary process, it is possible to consider whether there are patterns in internationalised software. And furthermore, whether there is value in combining these patterns with the Planet patterns described in this chapter. Chapter 11 discusses this possibility, and describes an extended version of Planet which addresses detailed software design.

Apart from the extended version of Planet, the work in this chapter has also led to another line of research. The research concerns the `Online Repository`, a concept central to Planet but also able to be generalised to other contexts. The next chapter describes how `Online Repository` can be generalised, as well as a software prototype of an `Online Repository` which has been developed in conjunction with this thesis.

Chapter 6

Reusing Knowledge About Users

6.1 Introduction

This chapter elaborates on the online repository introduced in Planet. The repository concept arose while creating the Planet pattern language, and was documented as a Planet pattern. However, it is a special pattern because it represents a way of reusing knowledge that is quite separate from that of patterns, i.e. reusing knowledge about users. This chapter therefore treats the online repository as an independent concept, another way of improving HCI design via reuse.

Users can vary in many different ways. Variables include Culture, as discussed in the previous chapter, as well as domain expertise, computer expertise, and organisational role. There are many techniques available for capturing information about users and their needs. One widely-known approach in industry is contextual analysis [20], which encourages usability specialists to observe users in their natural environment. In requirements engineering, there are techniques such as questionnaire interviews, open-ended interviews, and focus groups [90]. There are also more detailed design methodologies such as Jacobson's use cases [119], with its concept of an "actor", and Constantine and Lockwood's "user role models" [48]. Techniques such as these capture knowledge about users as well as requirements for a target system.

These techniques are all valuable ways to capture information for a specific project. However, it must be remembered that it is not always easy to obtain accurate user information. Users may be busy with other tasks, especially those in more influential roles such as managers and airline pilots. Devoting time to interviews may be regarded as time which could be better spent. These processes are also constrained by the resources of those specialists administering the study. Even when users are consulted, human nature is so complex that it can be difficult to predict how people will react to the new system when it is delivered.

If certain aspects of users can be classified in some ways, there is an opportunity to generalise results, so they may be reused for future work. Obviously there will be some loss of accuracy in comparison to repeating the study, but the time could be better spent refining the knowledge rather than reinventing the

wheel. The online repository embodied in the Planet language facilitates this kind of knowledge reuse.

In Planet, the form of user classification is by culture. Other applications of the repository might use different groups of users. For developers of organisation-wide systems, a convenient classification might be user roles, e.g. “data entry clerk”, “system administrator”. For a developer of a stockbroking product, user types might be “retail customer”, “stockbroker”, “auditor”. The repository should enable storage, retrieval, and reuse of knowledge about these user types.

This chapter expands on the online repository concept and introduces a repository prototype. The concept does not relate directly to the thesis aims, although it is, in some respects, an alternative means of achieving the first aim, the aim of reusing high-level design features to improve usability. Here, we are reusing knowledge about users rather than designs of previous systems. It is also important to expand on this concept due to its close relationship with the Planet language. Planet demonstrates how a pattern language can guide usage of a knowledge repository.

The chapter begins by providing an envisioned usage scenario for the online repository (Section 6.2). It then describes the general design goals for the repository and related web-based frameworks (Section 6.3). These goals led to the creation of the prototype online repository, and a walkthrough is provided (Section 6.4). Finally, there is a discussion of the prototype and the overall concept of an online repository of user knowledge (Section 6.5).

6.2 Envisioned Usage of an Online Repository

Design-level patterns like `Targeted Element` allow design-level concepts to be reused. However, there is also information about users that can be reused. One approach would be patterns of users, as speculated in Section 4.2.1.2. However, this sort of information does not seem particularly suited to the prescriptive format offered by patterns; it is too general in scope. Furthermore, there could not be much delegation from one pattern to another, so the possibilities of a coherent language are limited. The online repository provides an alternative way to declare knowledge about users, and continually update such knowledge. Following is an envisionment scenario to show how the repository and the pattern language work together.

David is a requirements engineer at Factory Expressware, an Australian company providing off-the-shelf productivity tools to manufacturers such as staff management software. The company has focused only on local customers to date, and has asked David to investigate the possibility of entering the Asian market with the existing Staff-Register program for performance monitoring and payroll management. David decides to create an internationalisation repository and apply the Planet language.

The starting point is to establish an `Export Schedule`. He explains to the marketing department and senior managers that the software development will progress more smoothly if they can provide any guid-

ance about future plans. After some deliberation, the marketing department gives him a tentative schedule: full Japanese and Chinese versions within twelve months, with other countries flagged as possibilities, at least for partial versions.

David looks at Staff-Register and, following the `Vector Metamodel` pattern, begins to write down dimensions of culture that he will be interested in. Some are the general overt and covert factors such as units of measurement and mental disposition. Other culture-varying factors are specific to the product, e.g. How are organisations structured? What variables are used to monitor staff?

After management sets up local agencies in the target market, David flies out and interviews the new staff to help him build up `Culture Models`. He also has access to some potential customers. In each country, he gathers information according to dimensions he is interested in. For instance, he visits a factory and looks at hard-copies of performance reviews to see what variables are considered. While at the factory, he borrows a computer and enters this information into the repository, which is available on the global intranet.

When David returns, he considers how the requirements must be changed to reflect what he has learned. First, he considers what new functionality is required (`Flexible Function`). Since taxation laws change the nature of employee payment, he refines the existing specification to provide more flexible payment arrangements. After looking at functionality, he moves on to the user-interface (`Elastic User-Interface`). He found that in some countries, managers were evaluating the performance of workers whom they did not know very well. “Usability will be enhanced by including a photograph of the worker,” he reasons, and he consequently specifies a user-interface which is “elastic” enough to show a photograph if desired.

Work continues over several months, with remaining patterns being applied and earlier patterns being revisited. The repository is continually updated as new information comes in. Eventually, the product is released, and management is happy with the result. At this point, the overall approach has helped David to structure his work.

However, it is a decision by management which brings out the full potential of the approach. Pleased with the initial work on Staff-Register, several other products are scheduled for export, including an accounting package and an inventory monitor. Much of the material about users and their domain has already been discovered. General issues like currency are known, as are details specific to the manufacturing industry. Some of this will not be directly applicable, such as which variables are used in performance reviews. Others will save a lot of work, e.g. information on taxation laws. The company will not have to re-obtain these details, and this is only part of the benefit. A more important advantage is that the Staff-Register program has already been implemented and tested in a real situation. If David hears complaints, he can update the repository accordingly. This allows undeveloped products to benefit directly from results of full-product testing. Thus, the problem of not being able to anticipate human needs becomes much less critical.

As for future versions of Staff-Register, these too will benefit. In expanding from English-only text to text in Chinese and Japanese, David could have required fonts only in those languages. However, the wiser choice of Unicode allows adaptation easily for other countries. If David used the `Export Schedule` effectively, he would have set in place flexible functionality and user-interface structures capable of adapting to the new cultures.

6.3 Online Repository: Design Goals

The envisionment scenario provides an overview of the nature of interaction which could occur with a repository. For this project, it was decided to develop a repository relating closely to Planet's goals. That is, a store of knowledge about overt and covert factors, for different cultures.

Many knowledge repositories in HCI and SE, such as design guideline documents, are static documents created by an authority organisation or "guru" individual, then consumed by developers (e.g. [7]). Even in an online environment, most knowledge repositories are static websites which only interact with their end-users via an email link (e.g. [209]). Acting on emails, if they are read at all, is completely at the discretion of the guideline producer. This narrows the scope of input and also imposes a difficult task on the maintainer. In some cases, there may be external means for users to influence a website, e.g. via working groups and mailing lists. However, this approach can be slow, invisible to most end-users, and difficult for individuals to contribute to. Moreover, it creates a large gap between the content and its maintenance procedure.

Fischer [80] has provided some views on HCI which are illuminating for the purposes of going beyond the prevailing model of static knowledge publication. He has observed that systems should be designed for ongoing evolution, and that domain workers do not fall in the category of novice users despite many systems catering only for novices. The online repository has been designed to support organic growth of knowledge about software and user culture. It is not just publicly readable; it is publicly writeable. Several features ensure users can take responsibility for the data on the site, with the maintainer acting only as a content moderator rather than content creator. When content is in the hands of the end-users, the system is capable of scaling up to store large, up-to-date, data. The web is an ideal medium to share information about cultural data, since it enables input from users and developers in any corner of the earth.

The concept of a dynamic site is also inspired by several existing website frameworks, in particular:

Dynasites [181] Dynasites is a software framework supporting the evolution of a shared information space. An example application is DynaGloss [180], an online glossary of technical terms. As well as reading the glossary, users can directly change it. They can add new terms, edit existing terms, link to related terms. A history of definitions for each term is also accessible. Dynasites is influenced by collaborative knowledge construction theory [182].

WikiWeb [228] WikiWeb is similar to Dynasites. The user community builds a website by adding, re-

moving and editing pages. There is one WikiWeb implementation [64] which is used extensively by the software patterns community to explore patterns and pattern theory.

FAQ-O-Matic [110] FAQ-O-Matic automates the time-consuming process of maintaining a Frequently Asked Questions (FAQ) list. A FAQ-O-Matic FAQ contains a list of "answers". Each answer is typically a question and a set of user responses. Users can create new questions and append to existing answer responses.

Slashcode [6] Slashcode is a framework enabling the webmaster to add news items and users to engage in discussion about each item. Unlike regular discussion forums, all active users can act as moderators by rating messages. Busy users can filter out low-rating information.

6.4 Working Prototype of an Online Repository

A web-based prototype [185] was developed as part of a Swinburne University summer internship program, following the high-level Planet patterns (Chapter 5). The prototype is called "Repository of Online Cultural Information" (ROCI). It consists of PHP modules, served by the Apache webserver, accessing a MySQL database, all running inside UNIX. The website can be accessed from a wide range of browsers, as it uses fairly standard HTML, with no reliance on client-side Javascript or Java. Some data has been entered into the repository, but the project is not ready for widespread usage at this stage (outstanding issues are covered in Section 6.5).

The site is focused on internationalisation issues, but the design could be generalised to user categories other than culture. It was decided internationalisation would be most appropriate for the present objective, (a) because it had been explored throughout development of Planet, and (b) because it is a particularly promising application of a repository publicly accessible on the world-wide web.

6.4.1 Locales, Factors, and Main Statements

Users access knowledge about "Locales" and "Factors" (Figure 6.1). The focal point of the site is the collection of "Main Statements", one Main Statement for each Locale-Factor pairing. The Main Statement is a concise summary of information for the pairing, e.g. Main Statement for Australia-Currency might be "Dollars and Cents have been used since 1966. One dollar is equivalent to 100 cents. $5.67 represents 5 dollars and 67 cents".

As well as the statement text itself, the page for a Main Statement also shows comments and user ratings (Figure 6.2). Users can respond to the statement through a form shown underneath it. They can submit a comment, a rating (from Strongly Disagree to Strongly Agree), or both at the same time. The comments are shown on the same page as the Main Statement. An average rating is calculated from all user ratings corresponding to the current statement.

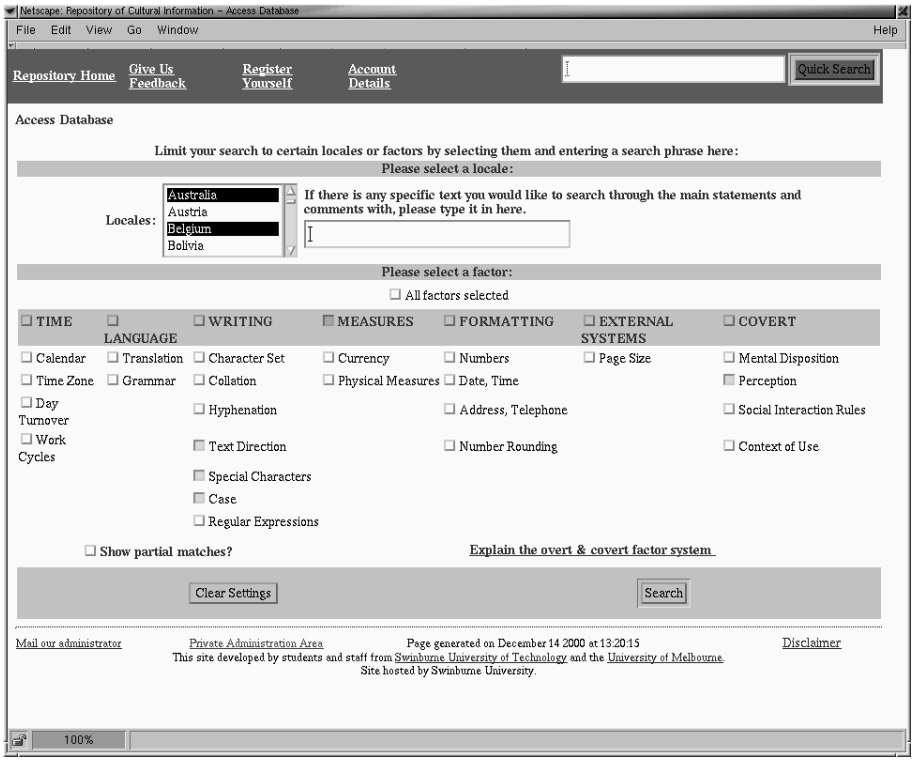


Figure 6.1. Online Repository screenshot: Browse/Search page. the user can search for a term via the top entry field. Alternatively, a number of locales can be selected as well as a number of factors.

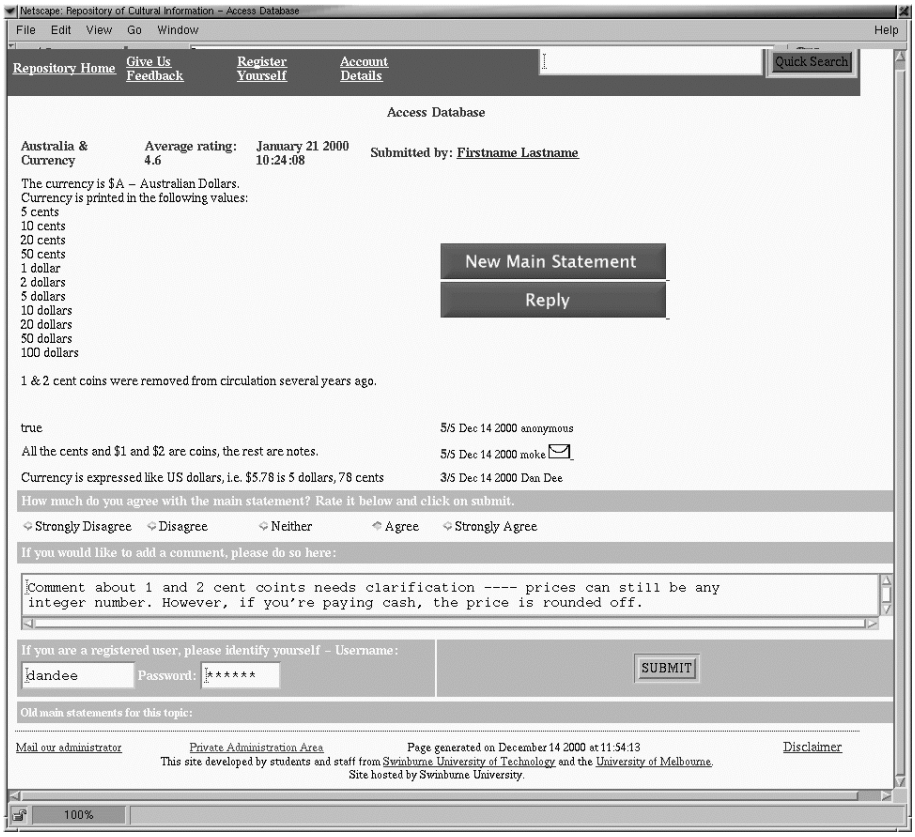


Figure 6.2. Online Repository screenshot: Main Statement Page. Current Main Statement is shown at top with invitation to change it. Below, user can submit a rating and/or comment.

Users can also submit a new Main Statement (Figure 6.3). When they do this, they can also specify any other locales and factors it applies to (Figure 6.4). It is common for a single statement to cover more than one locale, e.g. knowledge about the English language. At this stage, the repository is not sophisticated enough to directly link between Statements. Thus, repetition is required, but this facility at least allows the user to automate most of the repetition.

When editing a Main Statement, the intention is that the user will do this when the comments and ratings imply a change is warranted. Therefore, the comments are associated with a particular Main Statement version. Once a new Main Statement has been issued, the previous comments are no longer visible. Instead, there is a list of previous Main Statements. The comments corresponding to those Main Statements are linked from there (Figure 6.5).

6.4.2 User Contributions

When a website allows user contributions, there is an inevitable tension surrounding user registration. The obvious drawback with anonymous contribution is erosion of quality, with the site open to abusive statements and irrelevant commercial postings. This, in turn, places significant burden on the administrator. Furthermore, user registration can improve the community concept, with different users being known as authorities in certain areas. User registration also supports ongoing site improvement because it can facilitate tracking of user activity (subject to an appropriate privacy policy).

On the other hand, many users wish to remain anonymous. This may be due to privacy concerns or simply because they do not always feel like logging in. When a user encounters a statement which they feel they could contribute to, they will have to decide whether it is worth the effort. Forcing the user to log in at that stage necessitates further effort, and may tip the balance in favour of not bothering to contribute to it.

To deal with the tension, a two-pronged approach was taken. Users can register or stay anonymous, but registered users enjoy more privileges. Only registered users can add comments immediately . . . anonymous messages are placed in a “submission basket” to be approved by the administrator before going online. Only registered users can update the main statement. Both user types can enter ratings and these will be immediately updated.

It was stated above that the Main Statements are the focal point of the site. The user permissions reflect this; anyone can enter information, but only a more restricted set of users is permitted to alter them so as to reflect the comments. Anyone can register, so the restriction does not lead to the administrator having to maintain the statements. There is a requirement for the administrator to filter out unwanted messages. However, this should generally be a very quick task, because it should usually be quite obvious when the message is completely inappropriate. If the situation became out-of-hand, the program could easily be redesigned, either to prevent anonymous input or allow registered users to alter it. A more sophisticated approach is suggested by Slashcode (mentioned above), which lets active users rate messages.

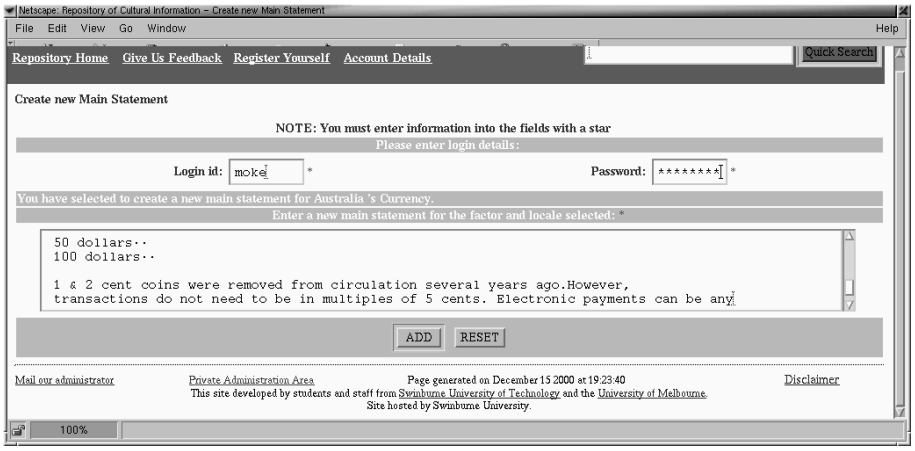


Figure 6.3. Online Repository screenshot: Entering new Main Statement (modifying Statement shown in Figure 6.2).

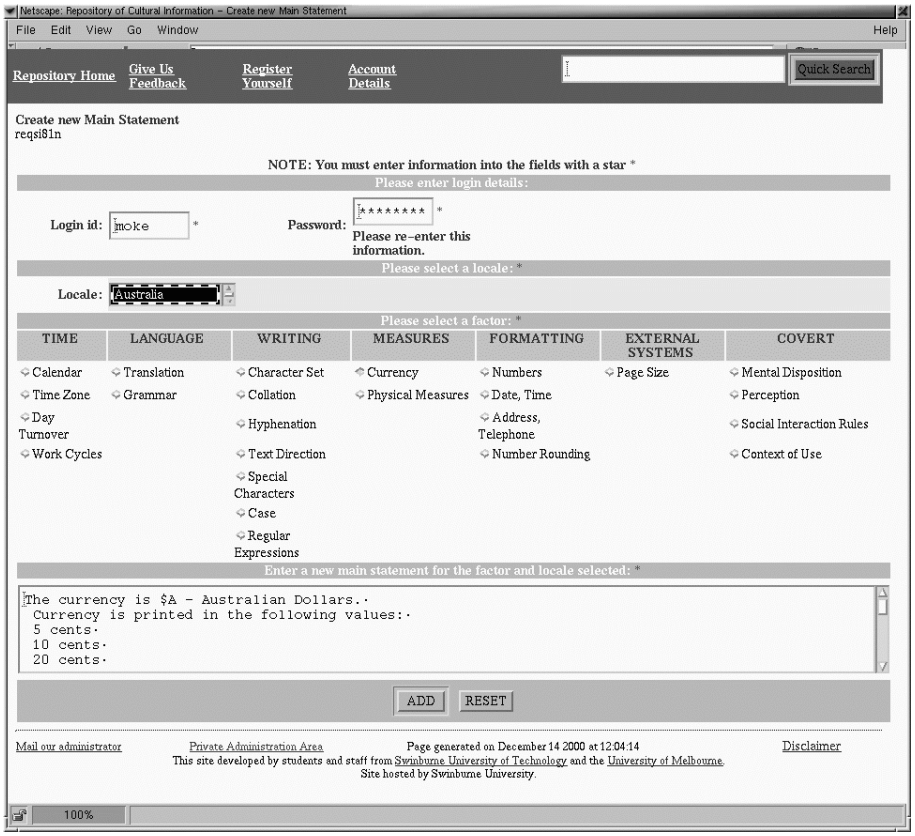


Figure 6.4. Online Repository screenshot: Specifying locales and factors belonging to new Main Statement. Users can specify that a single statement applies to more than one locale (follows new statement entry shown in Figure 6.3).

Figure 6.5. Online Repository screenshot: Repository updated after transaction shown in Figures 6.3 and 6.4. The previous Main Statement has now been moved to the list of old statements and the comments have been removed because they no longer pertain to the new statement. They can be accessed via the link in the old statement list.

6.4.3 Information Retrieval

Browsing the site consists of selecting one or more cultures and one or more factors (Figure 6.1). The user is then shown a summary of Main Statements for all combinations (Figure 6.6). From there, the user can visit the main page for the statement, the page which shows comments and history, and invites feedback. Users can further filter the results by entering a search term which results must match.

Some convenience controls are offered for users who wish to view multiple factors or locales. More specifically, it is possible to select a factor group, which equates to selecting all factors in the group. There is also an "All factors" setting that selects all factors in the program, and an "All locales" setting that selects all locales in the program.

The search form at the top of the site allows users to search through Main Statements and comments, providing a summarised list with links to further detail.

6.4.4 Administrator Functions

The major function performed by administrators is reviewing anonymous statements. For each statement, the administrator can approve, deny, or leave until later. There is also a provision to view recent posts by registered users, so that these may also be deleted if necessary (not currently implemented).

Other administration functions are: changing a user's password, removing a specific comment, and removing a specific user.

6.5 Discussion

The online repository concept is a powerful way to capture knowledge about users. A prototype, ROCI, has been developed to explore design issues surrounding the development of a repository, and a number of useful features have evolved during its development. In particular, the decision to centre discussions around Main Statements. This is similar to the way DynaGloss [180] centres around evolving glossary definitions. The inclusion of discussions may be related to Slashcode [6], in which discussions follow news items. In Slashcode, however, end-users cannot modify the news item. The WikiWeb forum on pattern languages [64] supports this notion by using the convention that people wishing to make comments append information. However, there is no explicit distinction between main content and user comments.

In ROCI, comments are explicit, and there is a history of statements. An older statement can be matched with the comments which specifically related to that version. A designer wanting immediate information should generally need to observe only the current Main Statement and any comments issued since it was generated. The Main Statement should reflect all comments leading to its creation. This is more efficient than having to traverse all comments since the site began, since many will now be out-of-date or redundant.

The current prototype implements the features which would essentially be required for an actively-

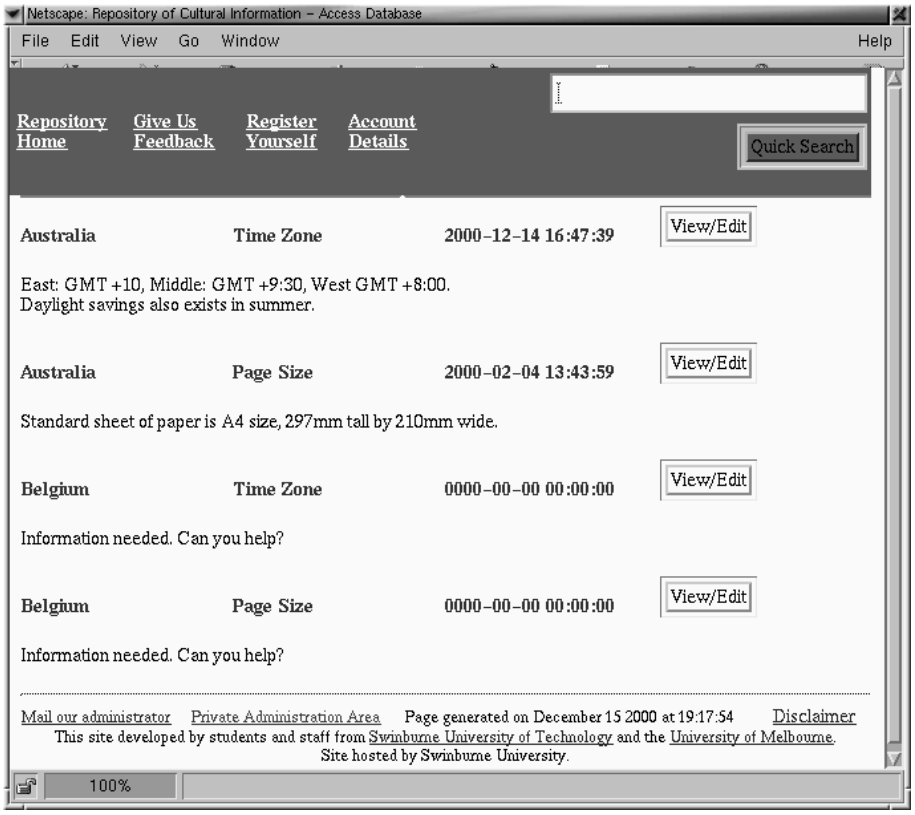


Figure 6.6. Online Repository screenshot: Main Statement summary. Following a search or browse request, the system shows the Main Statement for each Locale-Factor pairing, witb a link to the main page for that statement.

used and ever-evolving repository. However, it must be viewed as a work-in-progress at this stage. For practical reasons, a large degree of autonomy was given to the programmer. Thus, while most high-level information, such as that described in Section 6.4, was conveyed to the programmer, it was not possible to provide detailed information about the user-interface. To achieve a releasable system, at least two important enhancements would be required. First, presentation of user-interface, including text, would need to be more clear. Ideally, a task analysis would be conducted to optimise page flow and layout. Second, cookies should be used to provide a persistent session so that experienced users do not have to enter password each time they enter data. Third, the scoring system could be improved. Currently, there is no way for end-users to see how the average has been calculated; the only visible score entries are those accompanied by comments. Also, it would be useful to study how easily users can rate the statement; perhaps the ratings need to be more explicit than just ”Agree”, ”Strongly Agree”, and so on.

An online repository is a very different approach from design patterns, because it focuses on declaring knowledge about users rather than prescribing design strategies. The ideal situation is to combine the two approaches. Patterns alone cannot encapsulate a large knowledge base of this nature. Likewise, an online repository can benefit from usage instructions. The Planet pattern language provides guidance on using the repository, within the context of the overall internationalisation process.

Planet and the repository introduce the concept of high-level pattern languages for usability. It is also important to investigate how developers actually use pattern languages. The next chapter introduces a pattern language for safety- critical systems and describes a study in which subjects apply the patterns to a design problem.

Chapter 7

Safety-Critical Patterns: Theory and Empirical Study

7.1 Introduction

This chapter presents a second, domain-specific, pattern language. Like the Planet language of Chapter 5, it corresponds to the first aim of this thesis in that it addresses high-level HCI concepts. In this case, the key constraint is the specialised requirement of safety, meaning that a system failure can lead to a catastrophic event such as loss of human life or significant material damage. Many disasters have occurred, at least partially, because of poor usability. Examples of affected fields include medicine [134], aviation [94], and chemical engineering [126]. These problems impress the need to adequately treat usability, yet usability of safety-critical systems entails some complications which are not present in regular systems. For instance, should a system try to minimise task completion time when there is a risk a user will not spend enough time to think about the problem? A pattern language can show how designers have attempted to resolve tensions among usability principles.

The language provides a second example of a usability pattern language applied to a tightly-constrained domain, and also serves a second purpose. It has been used as the design tool for an empirical study whereby subjects were observed solving a design problem. The aim was to gain a qualitative understanding of the way people use patterns, and determine whether any changes should be made to the way the patterns are captured, organised, presented, and prescribed in practice. A two-group protocol was used, in which the first group received only high-level design principles while the second received the design patterns.

Section 7.2 outlines the methodology for discovering the patterns. Section 7.3 provides a background on usability in safety-critical systems. The Safety-Usability pattern language is described in Section 7.4. Section 7.5 describes the the experiment and suggests what the results imply for software developers and authors of pattern languages. Section 7.6 summarises the chapter and discusses opportunities for further

research.

7.2 Methodology for Pattern Discovery

As with Planet (Section 5.2), it is worthwhile noting the activities which were performed in creating this specific pattern language. The activities were similar to those in Planet, though there was somewhat more structure here with respect to generating the basic principles. The patterns were created in collaboration with Andrew Hussey at the Software Verification and Research Center, University of Queensland.

Identify Sub-Properties of Usability Popular style guides were summarised to arrive at a summary of usability properties. These were modified somewhat to reflect the objectives of this thesis. (This work [146] was performed with my supervisor prior to the safety-usability patterns research).

Identify Principles of Safety Literature on usability in safety-critical systems was used to arrive at a summary of principles for safety-critical systems. (This work [111] was performed by Hussey prior to the safety-usability patterns research).

Apply Properties to Safety-Critical Systems We considered how the five properties of usability must be adjusted to fit our concept of safety-usability. One sub-property — robustness — was considered to some depth due to its relevance to safety-critical systems. Its description is based on Hussey's safety-critical principles [111]. At the same time, we were also interested to address the relevance of traditional usability properties such as *task efficiency* to the safety-critical setting.

Obtain Case Studies Case studies were located in which the system at least partially supported safety-usability. For example, a system might provide feedback to the user in a manner which would improve safety. The sub-properties of usability are an explicit statement of the criteria we used to determine how appropriate these features are. The case studies were derived from research papers and practical experience of the authors. This is an area of difference with Planet, since international software is more readily accessible than safety-critical systems. Even though many safety-critical systems are documented, they are often documented because bad design has led to a disaster. Patterns naturally require examples of good design.

Group Similar Features The set of appropriate features which occur in the various systems, were informally placed into groups. Each group contained a family of system features which addressed the same sort of problem. Most of these groups evolved into design patterns. This was more formal than the Planet process, in which patterns were simply generated after observation. The grouping technique was a useful way to develop patterns when two authors are collaborating.

Document Pattern Relationships We looked for relationships among the patterns, and this led to the formation of a well-structured pattern language.

Review Patterns We each reviewed each other's patterns, and our supervisors later reviewed the patterns as well. The patterns were also reviewed during a submission to the Australian Workshop on Safety Critical Systems and Software [113], and feedback received at that workshop was also incorporated.

7.3 Usability in the Safety-Critical Domain

7.3.1 Sub-properties of Usability

Before looking at usability in safety-critical systems, it is important to provide a statement on general usability. In work performed prior to the development of safety-usability patterns, I analysed a set of style guides dealing with conventional windows-icon-mouse-pointer(WIMP) systems, texts such as Apple's Human Interface Guidelines [7]. The immediate objective was to produce a summary of desirable properties advocated by the style guides. A further objective was to modify this summary to produce a set of properties which would be relevant to less conventional systems, i.e. not just WIMP systems. The derivation of these properties is discussed in Appendix A.1 and summarised here.

Following is the list of desirable properties which arose from summarising the style guides.

- Consistency
- Familiarity
- Simplicity
- User Control and Feedback
  - Direct Manipulation
  - Flexibility
  - Feedback
- Visual Design
- Robustness
- Efficiency

The style guides relate to WIMP-based systems, but many safety-critical systems do not fall into this classification. Thus, the properties need to be broadened in order to use them in a safety-critical context. The properties which follow were adapted from those listed above, so as to fit a broader class of interactive systems.

Task efficiency (Simplicity, Efficiency) Task efficiency requires that software enables users of varied experience levels to accomplish tasks to the best of their ability. The property of simplicity is subsumed by this concept, but the concept also acknowledges that experts may require different interaction styles from novices. A smooth interaction enables users to achieve their goals without worrying about the idiosyncrasies of the application.

Reuse (Consistency) Whereas most standards focus on consistency, a preferable concept is reuse (borrowed from Constantine [49]). Consistency is a means to an end; its goal is to ensure that users can reuse the knowledge they have already attained. Given that a user can access many more documents and programs than when GUIs were first developed, it is apparent that a degree of diversity should be actively encouraged, lest all applications end up with an identical look-and-feel. This is becoming increasingly important as the number of applications rapidly increases. Similar applications may not only be cumbersome; they can also rob the user of vital cues required to distinguish between applications or between data items. Gentner and Nielsen [89] illustrated this point by noting that books on a bookshelf can exhibit a wide variety of appearances, but all are still recognisable as books. The diversity can actually aid the bookshelf "user" by providing a quick means of identifying particular books. If the general form of a book is preserved, then users are still able to reuse their existing knowledge base.

User-Computer Communication (Feedback, Visual Design, Familiarity) A direct-manipulation perspective on feedback says that the computer passively waits for user requests, processes them, and informs the user what has happened. This has led to distracting dialogue boxes (see Cooper [50]) and applications in which we see what we get (WYSIWYG), instead of what we need (WYSIWYN [49]). An approach which supports the goal of collaboration between the human and the computer is to ensure that the system facilitates communication. This means that the software should represents the system state in a way that is not only understandable to the user, but is also relevant to the task the user is performing. Thus, the interface represents changes to the system instigated by either the user or the computer. A rich communication environment should also enable the user to clearly express their needs.

Robustness (Robustness) Most user actions which programmers refer to as "mistakes" are not really mistakes to the user. Thus, while the concept of robustness has been retained from the summary of properties, there should be an emphasis on prevention rather than cure. Users should be free to explore, and, as Laurel [132] stated, "constraints should limit, not what we can do, but what we are likely to think of doing." (p. 105). When errors do occur, there should be easy paths to recovery.

Flexibility (Flexibility) The goal of user-computer collaboration is built upon the idea that characteristics of both human and computer must be considered. Flexibility ensures that these characteristics are

taken into account. Providing recordable macros, for instance, is a good way to reduce a human's workload. Enabling the computer to reduce accuracy as a means of speeding up the interaction would be an example of accounting for the computer's characteristics. In keeping with the goal of collaboration, it may be best to enable the computer, the human, or both, to make such changes.

Comprehensibility (Simplicity, Familiarity, Visual Design) Comprehensibility denotes that the interface should be at the right level of detail so as to be comprehensible to the user in question. As with Task Efficiency, this property acknowledges that not all users are beginners, and a "comprehensible" interface is therefore a more useful goal than a "simple" interface. Visual design is covered by this term too, because it should follow from an understanding of what is comprehensible to the user.

7.3.2 Distinguishing Features of Safety-Critical Systems

The properties in Section 7.3.1 refer to a conventional view of usability. But safety-critical systems impose new challenges for usability practitioners, a theme which is explored in this section. Following are several ways by which safety-critical systems differ from conventional desktop applications (e.g., word-processors, browsers), and the implications of this difference for safety-usability:

- Aspects of usability relating to robustness and reliability (as measured by failures, mistakes, accidents, etc.) take preference over other usability attributes such as efficiency, reusability of knowledge (learnability and memorability), effectiveness of communication and flexibility, relative to their importance in non-safety-critical systems. Of course, these other attributes are still important, especially where they enhance or have a neutral effect on safety.

- Safety-critical systems often involve monitoring and/or control of physical components (aeroplanes, nuclear reactors, etc.). These systems usually operate in real-time. Another implication is that physical interaction environments may be quite diverse, and a model of the user quietly sitting at their desk is often inadequate. Furthermore, the systems may also be distributed across many users and/or locations, e.g., a plane trip involves co-ordination between the plane itself, its start and end locations, as well as other planes and flight controllers along the way.

- Safety-critical systems often involve automation. Cruise control and auto-pilots are well-known examples. Automation alone is not sufficient to remove human error. Bainbridge describes the following ironies of automation [10]:

  – Designers may assume that the human should be eliminated due to unreliability and inefficiency, but many operating problems come from designer errors.

  – Automated system are implemented because they can perform better than the operator, yet the operator is expected to monitor their progress.

  - The operator is reduced mostly to monitoring, but this leads to fatigue, which in turn reduces effectiveness.
  - In the long-term, automation reduces physical and cognitive skills of workers, yet these skills are still required when automation fails. In fact, the skills may be in more demand than usual when automation fails, because there is likely to be something wrong. De-skilling also affects workers' attitudes and health adversely.

- Users are generally trained. Although the user interface should be as intuitive as possible, it is usually not necessary to assume novice usage will take place. This contrasts with the principles which might be required for a mobile phone or a web browser. Because users are trained, the diminished flexibility that may arise from enhanced robustness is of less consequence than would be the case in a conventional setting.

- Likewise, it is assumed that safety-critical projects are performed with a high consideration for software quality, to achieve robustness and ensure that the system functions as it is designed to. Developers should be highly qualified and the process may allow for steps which might not normally be considered, such as parallel design [172] and formal methods.

7.3.3 From Usability Properties to Safety-Usability Properties

In light of the previous section, the usability properties of Section 7.3.1 are now revisited. The purpose is to ask whether they are still relevant to a safety-critical context, and to determine how each principle should be attuned. The properties are important because the patterns are supposed to reflect the underlying principles, as described in Chapter 3. This helps to fulfill the goal of reusing high-level design which is fundamental to this thesis.

Each property is shown below, with a summary statement, and an explanation of its implications for safety-critical systems.

Robustness *Errors should be rare, recoverable, and their effects as harmless as possible.*

Implications: In a safety-critical context, robustness is definitely the most important of the identified usability properties. The earlier definition of robustness remains valid.

Based on the literature, Hussey [111] has suggested three high-level principles for achieving robustness: error prevention, error reduction, and error recovery. For error prevention, unsafe states should be anticipated and prevented. Error reduction can be reduced with *forcing functions* [176], clear displays, ensuring user alertness, and enabling reasoning by the user. Error recovery is improved when the user can recognise problematic states, when the system explicitly warns the user of such states, and when the user can request state data from the system.

Task Efficiency *Help users of varied experience levels to minimise effort in performing their tasks.*

Implications: Throughput should generally increase if fewer errors occur, suggesting error reduction and prevention techniques can enhance task efficiency. However, introducing these features can also come at a cost if they impact on user interaction. For instance, redundancy-based error reduction may require the user to enter the same information twice. Such a technique would have a detrimental impact on task efficiency. On the other hand, consider a user who must enter a four-digit ID code. An entry box might provide an afforance showing that exactly four digits must be entered. This would reduce the chance of an incorrect ID being entered, and, if well-designed, need not have any negative impact on task efficiency. In summary, techniques for error reduction and prevention may have a positive or negative influence on task efficiency; the influence depends on the extent of interruption the techniques have on the user's work.

Reuse *Ensure that users can reuse existing effort and knowledge.*

Implications: The safety requirement means that techniques for reuse — such as consistency — can be a double-edged sword, just like the interaction with task efficiency. Safety can be improved by making routine actions become subconscious so that users can concentrate on higher-level issues [166]. However, reuse of previous erroneous inputs and reuse of actions from one task in another task can also be a source of error. Reuse can circumvent error reduction techniques such as redundancy-based checks, since users may automatically confirm actions without considering the situation. Reuse implies that two different sequences may contain a common sub-sequence, and this cause capture errors. Capture errors occur whenever two different action sequences have their initial stages in common, with one sequence being unfamiliar and the other being well practised. For example, intending to shut the system down, a user may access the "System Operations" menu. Once there, the user may inadvertently select a frequent operation such as "Adjust lighting". This continuation of an incorrect sequence is caused by its opening actions being identical to the less frequent sequence, a consequence of designing for reuse.

User-Computer Communication *Facilitate collaboration between humans and computers by appropriately representing changes to the system (which have been instigated by humans or computers).*

Implications: Real-time systems are dynamic, and frequent state changes can confuse users. The representation should account for such variances, and show only those changes that are necessary. An example of an appropriate representation is a car speedometer, which is analogue because a changing digital meter would be unreadable [178]. The physical environment should also take into account human input and output needs. Leveson cites an example where this principle was clearly not applied: a nuclear power plant where controls are placed thirty feet from the panel meters, yet users must be within a few feet to read the display [134].

Flexibility *Account for users of different backgrounds and experience by allowing for multiple ways to perform tasks and for future changes to the system requirements.*

Implications: Error prevention techniques such as forcing functions essentially introduce additional modes into a system. Moded systems are typically less flexible: the user must follow the order of actions to perform the task that was envisaged by the designer of the system. System qualities must be traded off against each other. User-initiated flexibility is often *undesirable* for safety-critical systems, because frequent changes to the user-interface may lead to errors if users forget the change was made, or if one user switches to another's environment. Rapid switching between interface devices are not uncommon during emergency situations.

7.4 The Safety-Usability Pattern Language

This section provides an overview of the safety-usability pattern language, explaining the patterns and their relationship to each other.

7.4.1 Structure and Overview

The patterns are all concerned with reducing user error or the consequences of user error. The structure of the language is illustrated in Figure 7.1. As the figure shows, the patterns are grouped according to four aspects of the human-computer interaction:

Task Management The control flow of the human-computer interaction.

Task Execution The physical mechanisms by which users perform tasks.

Information The presentation of data to users.

Machine Control Actions and intervention by the machines which complement and compensate for user capabilities.

Following is a brief summary of the problem addressed by each pattern and the solution provided. The full text of patterns may be found in Appendix C and a sample of the full-text patterns are shown in Section 7.4.3.

7.4.1.1 Task Management Patterns

Stepladder: Where systems require operators to perform complex tasks it is preferable to split the tasks into a chain of simpler tasks.

Recover: If the system allows users to place the system in a hazardous state, then users need a facility to recover to a safe state.

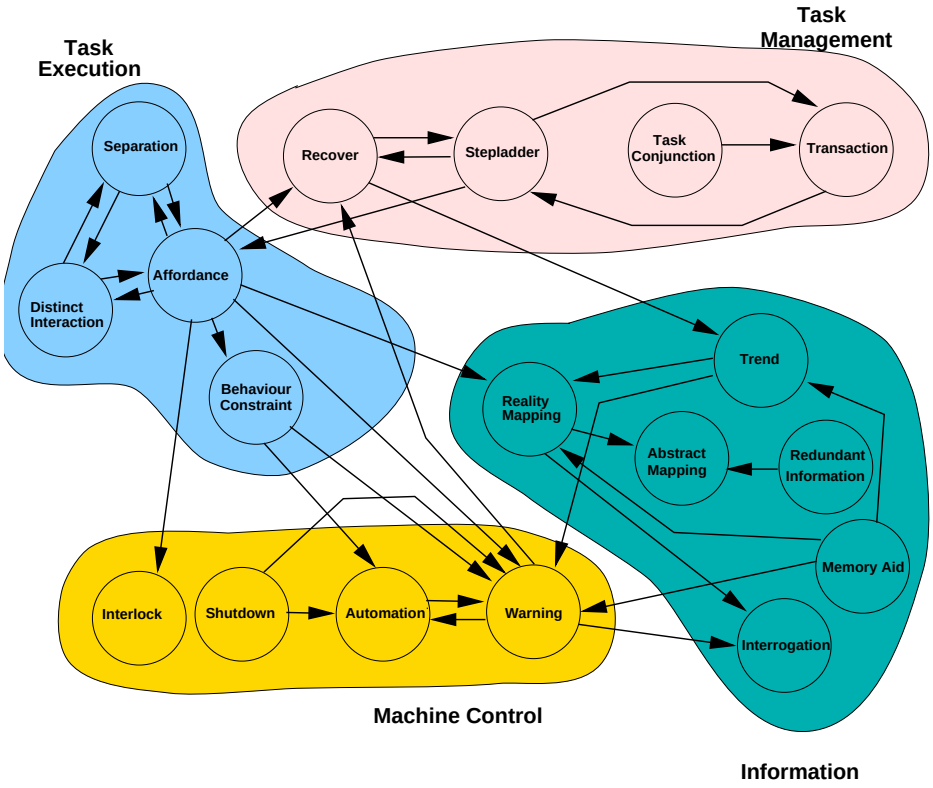


Figure 7.1. A map of the Safety-Usability Pattern Language. Patterns and pattern groups within the language. Arrows indicate process flow, i.e. the Resulting Context of the source pattern suggests the destination pattern may be used to solve a problem arising from the application of the source pattern. The clusters represent four groups: task management patterns, task execution patterns, machine control patterns, and information patterns.

Task Conjunction: The system can ensure that a user's action matches their intention by checking for consistency and by requiring that the user either repeat tasks, or perform multiple simultaneous actions (e.g. multiple-key combinations) to perform tasks.

Transaction: The need for recovery facilities in the system is lessened if related task steps are bundled into transactions, so that the effect of each step is not realised until all the task steps are completed and the user commits the transaction.

7.4.1.2 Task Execution Patterns

Separation: Components of the user-interface should be physically or logically separated if they are used in a similar way and for which incorrect operation could be hazardous.

Distinct Interaction: Components of the user-interface should be operated by distinct physical actions if they could be confused.

Affordance: User-interfaces should contain features which suggest how they should be used. An example of a design affordance is a slider enabling a magnitude to be selected. If presented appropriately, the slider's appearance should help the user to appreciate that the slider can be moved to alter the magnitude.

Behaviour Constraint: The system should prevent users from requesting hazardous actions by anticipating such actions and allowing only those that are safe.

7.4.1.3 Information Patterns

Reality Mapping: To facilitate user understanding of the state of the system, the system should provide a close mapping to real-world objects where possible. This mapping should be supplemented with abstract representations that enable a rapid assessment to be made where necessary.

Abstract Mapping: Where reality mapping is infeasible, unnecessary to ensure safety, or, indeed, not able to capture the essence of complex data with clarify, alternative representations should be used to facilitate rapid assessment of the system state. Such mappings summarise the system state, providing a small number of salient data points for monitoring, rather than a large collection of raw data.

Redundant Information: Where information presented to the user is complex or could be misinterpreted, provide multiple views so that the likelihood of error is reduced. Such redundant views should be observably consistent.

Trend: Humans are not good at monitoring, so when the system state changes, the system should compare and contrast the current state with previous states.

Interrogation: Presenting the entire state of the system to the user would be overwhelmingly complex in many cases. The user should be presented only with the most salient features of the system state, but be able to interrogate the system for more information where necessary.

Memory Aid: If users must perform interleaved tasks, memory aids should be provided for recording information about the completion status of tasks.

7.4.1.4 Machine Control Patterns

Interlock: Interlocks provide a hardware-based way of detecting and blocking hazards, so that even if errors occur, they cannot lead to harmful outcomes.

Automation: If a task is unsuitable for human performance, because it involves continuous or close monitoring or exceptional skill or would be dangerous for a user to perform, then it should be automated.

Shutdown: When shutting down the system is simple and inexpensive, and leads to a safe, low risk state, the system should shut down in the event that a hazard arises.

Warning: To ensure that users will notice hazardous system conditions when they have arisen and take appropriate action, warning devices should be provided that are triggered when identified safety-critical margins are approached.

7.4.2 Case Studies

Several case studies were particularly instructive in forming these patterns. The case studies are detailed in Appendix B.1, and are outlined here:

Oil Pressure System Two dials show fluid levels in an aeroplane for rudder and aileron reservoirs (from Fields and Wright [79]).

Hypodermic Syringe A medical syringe with dosage entered by keypad (from Dix [71]).

Druide An air-traffic control system using an overhead map to display location of aircraft and related data (from Palanque [183]).

Railway Control A system for co-ordinating and controlling the movement of trains (relates to work conducted by Hussey and colleagues at the Software Verification Research Centre).

As the case studies demonstrate, examples from a diverse range of safety-critical contexts were sought.

7.4.3 Examples of Full Patterns

Full text for all patterns may be found in Appendix C. This section contains four representative patterns drawn from the language, one pattern for each group.

Pattern 7-1. Transaction [Task Management Pattern]

Context: Actions are not time-critical and hence can be "stored-up" before being applied and:

- Sub-steps can be undone;
- The effect of the task as a whole is difficult to undo;
- Risk is relatively low compared to cost and difficulty of prevention.

Problem: For many real-time systems, it is difficult to provide a `Recover` action which has any practical value, because the system usually changes irreversibly by the time the user tries to recover from the unwanted action.

How can we improve recoverability?

Forces:
- It is relatively easy to recover tasks that do not impact on real-world objects.
- Often, reversal is useful to iteratively define a task's parameters.
- Transactions are used in data-processing to enable the effect of a sequence of actions to be "rolled-back" to the state at the commencement of the transaction.
- Transactions bundle a sequence of task steps into a single task, hence they are ideal for structuring interaction in terms of overall goals and sub-tasks.

Solution: Bundle several related task steps into a transaction, such that the effect of each step is not realised until all the task steps are completed and the user commits the transaction. By grouping task steps in this way, it becomes very easy to `Recover` the effect of a sub-step before the transaction as a whole has been committed. In addition, because errors are deferred until the transaction is committed, users have more time to consider their actions and to recover from them if appropriate.

Each transaction should involve task executions and information that is physically grouped on the users console or display. For example, a data entry transaction might be implemented as a pop-up window with commit and abort buttons to either accept or reject the information entered by the user.

Examples: In the Druide system, messages to aircraft are first constructed, and then sent to the aircraft using the "SUBMIT" button. The user can cancel a message before they submit it. A similar facility is available in some email systems which actually hold mail for several minutes before sending it.

A standard dialogue box supports this pattern. The user can easily enter data via text fields and buttons, but none of these choices matter until they hit a confirmation button (e.g., one labeled "OK").

Resulting Context: Task steps are grouped into transactions with commit and abort options for each group. If it is appropriate for the transaction's sub-tasks to be constructed iteratively, then the transaction can be viewed as a form of `Stepladder`. A sequence of transactions themselves can also form a `Stepladder`.

Pattern 7-2. Behaviour Constraint [Task Execution Pattern]

Context: It is possible to determine system states where certain actions would lead to an error.

Problem: The most direct solution to improving assurance through diminished user error is to prevent users from making errors.

How can we prevent users from requesting undesirable actions?

Forces:
- Even if we provide appropriate information and guide users by providing `Affordances`, users will inevitably make some slips and mistakes.
- The system has a large amount of data available concerning the state of objects.
- In some circumstances, designers will be aware that certain tasks will be undesirable in certain situations.
- It is preferable to trap user errors before they impact on the system, rather than detecting potentially hazardous system states and then attempting to rectify the situation.

Solution: For any given system state, anticipate erroneous actions and disallow the user from performing them. The idea here is to prevent the action from occurring in the first place, rather than dealing with whatever the user does. The logic could be programmed directly, e.g., "If the plane is in mid-air, disable Landing Gear button". It could also be implemented via some intelligent reasoning on the machine's behalf, which would require the machine to understand what tasks do and where they are appropriate.

This is a very fragile approach which should be used with extreme caution. It assumes that the condition is measured accurately and that ignoring the constraints always implies a worse hazard than following them. It is therefore risky in unforeseen circumstances. This can be somewhat alleviated by a mechanism allowing an appropriate authority to override the constraint.

A further disadvantage is that this pattern leads to a less flexible user-interface (i.e., the interface is moded) and the user may become frustrated with the structured form of interaction if they do not properly understand the tasks they are performing. In a safety-critical domain, this should be a less severe problem than normal, because the user should be highly trained.

Behaviour constraints are usually implemented as an additional system mode, e.g., by "greying out" menu items or buttons where appropriate. As such, behaviour constraints require close automatic monitoring of system state and therefore a frequently used partner pattern is `Automation`.

Examples: Kletz [126] gives everyday examples, such as components that will only fit together in one way, making incorrect assembly impossible. Norman [177] also gives such everyday examples, such as fire stairs that have a rail blocking access to the basement stairs; the error of proceeding all the way to the basement is prevented. In the Druide system, users are only given access to menus to change the system state, preventing errors associated with entering invalid data. In the Railway Control system, users cannot direct a train to move to an occupied section of track.

Resulting Context: Some user errors are no longer possible. The system will usually be more heavily moded than prior to applying the pattern. The system may inform the user that a requested operation is unavailable (i.e., applying the `Warning` pattern).

Pattern 7-3. Trend [Machine Control Pattern]

Context: Users need to formulate and follow task plans that involve attention to the change in state of the system, e.g., where an action must occur if the state is in a certain configuration, or when a state change occurs.

Problem: How can the user be notified that the system is approaching dangerous territory?

Forces:
- Many user errors stem from memory limitations. Users may not notice that the state of the system has changed and that they should take action;
- Memory-based errors may occur even when the user has previously formulated a plan to perform a particular action when the state of the system reaches a particular configuration. For example, in air-traffic control, the user may need to change the altitude of an aircraft before it reaches a particular waypoint but may not be able to do so immediately because of other more pressing concerns; a hazard arises when the user fails to return to the original aircraft and change its altitude, after resolving the immediate concern.

Solution: Allow the user to compare and contrast the current state with previous states. This will help users assess the current situation and formulate plans for the future.

Examples: The Druide system displays aircraft as a trail of locations, with the most prominent location displayed being the immediate location (see Figure B.4). The Oil Pressure system displays oil pressures in the left and right ailerons and a shaded region that indicates the oil pressure in the previous 5 minute interval (see Figure B.1).

Design Issues: One common technique is to overlay previous states on the same display, showing the previous state in a less visually striking manner (e.g., muted colours). This is particularly effective for motion, as a trail of past motion can be formed. If this technique causes too much clutter, a replica of the past state can be placed alongside the current state. This, however, occupies valuable screen real estate, and may hamper direct comparison.

Resulting Context: State changes are explicitly displayed to the user. Display of the state change is a `Reality Mapping`. The change in state may also be brought to the users attention via a `Warning`, if it has a readily identifiable safety implication.

Pattern 7-4. Interlock [Machine Control Pattern]

Context: Risk is sufficiently high that measures to block the occurrence of error do not give sufficient assurance and the bounds of acceptable system outputs can be defined.

Problem: How can we be sure that errors cannot lead to high risk hazards, even if they occur?

Forces:
- `Behavioural Constraints` may not prevent all incorrect commands because systems are too complex to predict all possible states and events.
- Measures to diminish user errors should not necessarily be regarded as sufficient evidence of system safety and additional evidence may be required if risk is sufficiently high.

Solution: Anticipate errors and place interlocks in the system to detect and block the hazards that would otherwise arise. Interlocks can be embodied in hardware or software, preferably both. But there is no point creating an interlock if the system failure causes the interlock itself to work incorrectly.

Examples: Many modern motor cars come equipped with Anti-Lock Braking Systems (ABS). Such systems are interlocks, screening aberrant driver behaviour. If the driver presses too hard on the brake pedal, the ABS will override the driver's actions and maintain brake pressure at an optimum rate. The Therac-20 and Therac-25 machines are medical equipment designed to administer radiation doses to patients [133]. The Therac-20 machine has a hardware interlock that prevents extreme doses of radiation but the Therac-25 machine does not. The Therac-25 machine was involved in several well-publicised accidents that arose because of an error in the software which failed to correctly update the dose after the user had corrected it on screen. Detection of errors assumes that the hazard situation can be formulated in a straight-forward and error-free way. As an example in which this was not so, consider the Warsaw accident in which A320 braking was prevented because braking logic required both wheels on the ground (e.g., see [133]). These issues are considered further in `Automation`.

Design Issues: If a system is designed using only interlocks to prevent hazards arising from user error, removal of the interlocks opens the system to the possibility of hazardous operation. Interlocks therefore should always be used with `Behaviour Constraint` and `Intended Action` to provide 'defence in depth'.

7.4.4 Comments on the Safety-Usability Patterns

As Section 7.3 discussed, our notion of usability must undergo some attunement in the context of safety-critical systems. Designers must struggle with many of the regular concerns of usability, and simultaneously ensure that hazards will be rare and of low consequence. In the world of safety-critical systems, there are numerous post-mortems of systems gone wrong. Many contain valuable lessons for usability practitioners, and have certainly been acknowledged within the present work. However tempting it may be to retrospectively criticise poorly-devised systems, there remains a great need to establish in positive terms how design may occur.

The principles of Section 7.3 provide some direction for designers. However, it is much easier to state broad principles than to show what they mean in design contexts. Especially in the context of safety-critical systems, where there is substantial conflict between robustness and other aspects of usability. The patterns above are motivated by the need to expose concrete, real-life, solutions to specific problems.

The patterns generally achieve their goal to this end. At an individual level, each pattern generally shares one piece of wisdom which is backed by real-life examples. One pattern, `Affordance`, is in retrospect, rather broad and is really more principle than pattern. However, this seems to be the exception, since the remaining patterns have specific contexts and problems, and show how to juggle the conflicting safety-usability forces.

The patterns have evolved into a structure quite different from Planet (Chapter 5). As a language, Safety-Usability Patterns come close to Planet, but are slightly less-connected overall. Planet is a more hierarchical structure, and all patterns are drawn together by several central patterns — `Multicultural System`, `Culture Model`, `Online Repository`. The safety-usability patterns are in some ways four mini-languages, one for each grouping, since there tends to be tight integration within each group and less to external patterns.

The tight integration is demonstrated by considering the following thread of `Task Management` patterns. A designer worried that an unintended action will lead to a fatal situation can employ `Conjunction`, forcing the user to execute the task in two different ways. This `Conjunction` task may typically be performed in a sequence of related tasks, leading to the construction of a `StepLadder` which will help the user step through the sequence. The strength of integration is similarly strong among the other groups. Across the groups, there are certainly valuable interactions; for example, the advice to help the user understand the effect of `Recover` by showing a `Trend`. Thus, the language is something like a network of inter-weaving mini-languages rather than a hierarchical set of patterns in the style of Planet.

As with Planet, the specialised requirements are tightly-scoped. Once again, it is apparent that constraining this dimension has enabled the consideration of constructs at different levels — Task Management, task execution, machine control, and presentation. While the integration is closer within these groups, there is still a rich interaction across patterns at different levels of abstraction. This stands in contrast to a generic attempt at a usability pattern language, which would probably be unable to tie these concepts together in a coherent manner.

7.5 Experiment: Designers applying Patterns

There have been several case studies on patterns in industry (e.g. [16, 33, 128, 41]; see Section 3.2.2). However, little effort has been made to study patterns in a controlled experiment. The closest known work concerns pattern-aided design documentation — Prechelt [188] asked whether design documents become more clear if their authors explicitly indicate where patterns have been used. The present study is significantly different because it addresses the issue of designing according to patterns, and also because it applies to HCI patterns rather than detailed software design patterns such as the Gamma et al. [88] patterns used by Prechelt. The present experiment is a controlled study on application of patterns to a design problem.

It was important to have a number of participants so as to ensure diversity of personality, since patterns, as with any design technique, are likely to be used in different ways by different people. To achieve this goal, given constraints of subject availability and administrator time, it was necessary to recruit subjects for sessions of a few hours rather than performing one or two projects over a period of days or weeks.

Another critical decision was to use a two-group methodology: one group being introduced to patterns, the other group being introduced to a more generic design technique. This allowed the effects of patterns to be more accurately gauged.

With sessions lasting only a few hours, it was not feasible to perform a quantitative comparison in any meaningful way. Instead, the experiment was designed with the aim of capturing as much rich information as possible about how people use patterns. In turn, it was anticipated that such observations would lead to practical advice for developers and pattern authors. A secondary aim was to provide at least some insight into the relative strengths and weaknesses of the patterns (such as those described in Section 3.2.3), even if the short-term nature of the study does not allow a comprehensive analysis.

Safety-critical patterns were chosen in preference to Planet as a basis for this study. The major reason is that they are more suited to a short time period. Planet has a broader scope, addressing organisational issues and the articulation of culture models, which renders it impractical for a short study.

Even though safety-critical patterns are more directly applicable to a design problem, there was still a concern about the allocated time. Subjects were required to learn about patterns in general, then safety-usability patterns, then apply the patterns. Instead of starting with a new problem, subjects were provided

with a specification of an existing system and asked to improve it. This helped to accelerate subjects' understanding of the problem.

7.5.1 Experimental Design

7.5.1.1 Subjects

The subjects were sixteen students at the Computer Science and Software Engineering Department at the University of Melbourne. They were on a summer studentship program at the time, and most were about to enter their final year of undergraduate studies. Two subjects (the "Patt-D" pair) had previously been exposed to software design patterns; the rest were unfamiliar with the patterns concepts.

7.5.1.2 Design

Subjects performed the design task in teams of two, collaborating to arrive at a final solution. Open communication between the partners was encouraged, enabling the administrator to gain insights as to the thought processes occurring during design.

Four pairs used a patterns-based approach (labelled "Patt-A"..."Patt-D") and four pairs used a principle-based technique ("Princ-A"..."Princ-D"). An example problem was presented first followed by the real problem. In both cases, subjects used their assigned design approach to solve the problem.

7.5.1.3 Materials

The materials provided to subjects may be found in Appendix D. Following is a summary (excluding ethics-related materials):

Instructions Experiment overview, introduction to design techniques, expected outputs. Two versions were used; one for each group.

Design Tools Design principles for robustness, task efficiency, reuse, user-computer communication, based on the principles in Section 7.3[1]. Subjects using the patterns were also provided with the principles, and in addition, were provided with the complete set of patterns, an overview of the patterns, and descriptions of the accompanying case studies.

Example Case Study A factory-based alarm system allowing monitoring of sensors.

Real Case Study An application for radiologists who are performing laser emissions into the head. Certain aspects were simplified in the study (e.g. the time interval between laser beams was ignored, the possibility of firing from two different heights was disallowed).

The major issues in the case study are:

[1] To avoid subject confusion, flexibility was removed (the patterns themselves rarely address flexibility).

- A patient needing treatment has their head fixed in a stationary position. Then two lasers, coming from various angles, each emit a beam. The radiologist aims to make the lasers meet at the area in need of treatment. At the intersection area, the energy is highest because it is the total of the energy of each beam.
- The hardware consists of two lasers on a circular track (Figure 7.2). Each laser can move 360 degrees around the track. The radiologist controls the following parameters: θ (0.0-360.0 °), the angle around the track; γ (0.0-360.0°), the angle indicating the direction of the laser; *w*, (0.0-100.0 μm), the beam width; *t*, (0.0-600.0s), the time the beam is emitting.
- Several usage guidelines were provided, e.g. patient's head should not move, heart-rate should be monitored, beam intersection is an *area* rather than a *point*[2].
- The initial design sketch consisted of an overhead diagram of the patient's head, and lines representing where the laser will be fired (Figure 7.3). The radiologist can set the hardware parameters for the beams, switching between beams via radiobuttons. The system provides some status information during beaming (Figure 7.4).

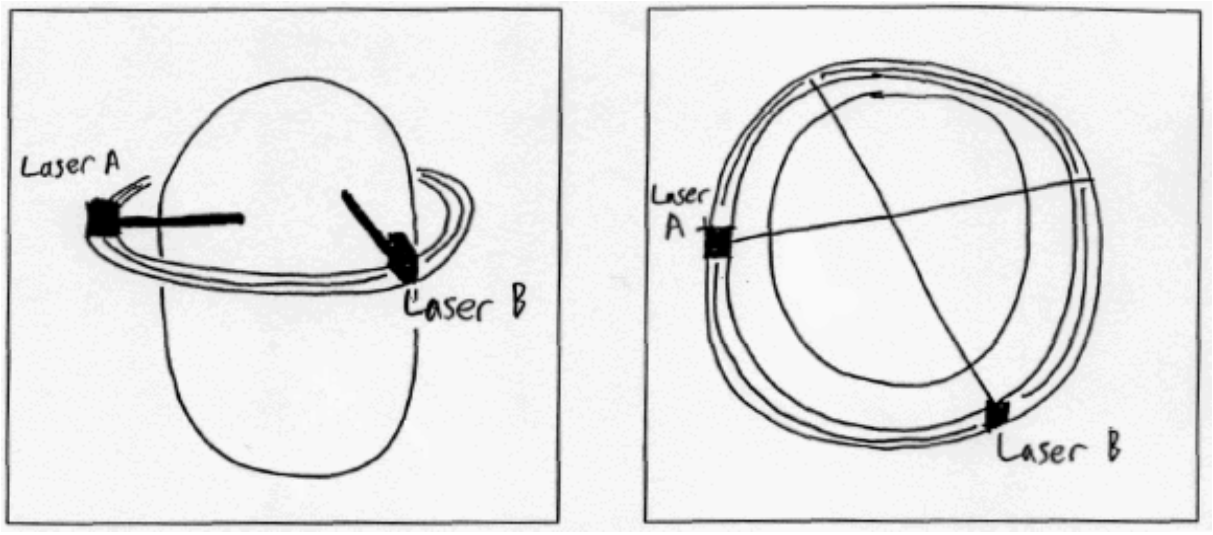


Figure 7.2. Safety experiment case study: Hardware setup, showing lasers on a circular track which surrounds the patient's head. (a) perspective view, showing circular track around a slice at a fixed height. (b) top view.

7.5.1.4 Procedure

Subjects were informed that the researchers were assessing the applicability of a design technique and wanted to observe people using the technique. As background material, a brief definition of usability was

[2] Technically, beam intersection is a volume, but this study only considered a 2-dimensional cross-section.

Figure 7.3. Safety experiment case study: Hand-drawn sketch of initial prototype during planning phase

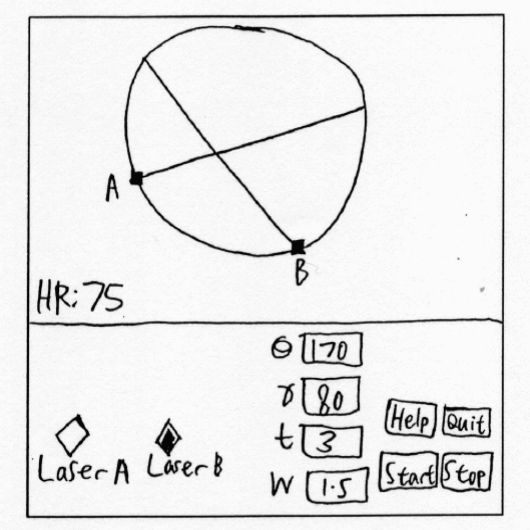
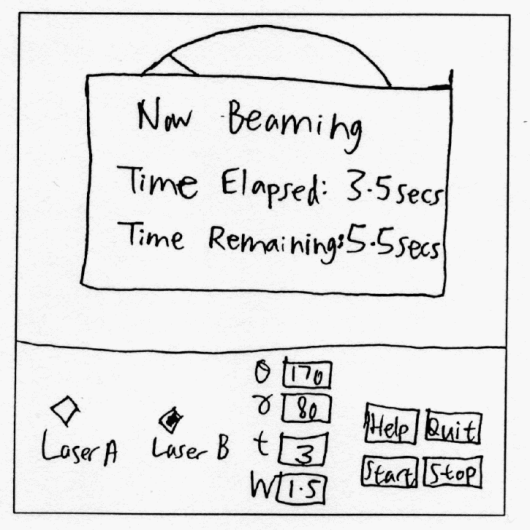


Figure 7.4. Safety experiment case study: Hand-drawn sketch of initial prototype during beaming phase



given.

Subjects were then provided with the design material: a list of principles for the principles groups, or a list of principles, and a description of the patterns and accompanying case studies for patterns group. They were able to spend 10-15 minutes reading the material. Following this, they were provided with the example problem. After the example problem, an example solution was presented which showed how some patterns might be used. This solution was shown to both groups, though references to patterns were only given to subjects in the patterns group. After a short break, subjects performed the real problem and then a semi-structured de-briefing interview was conducted.

For both example and real problems, subjects were allowed as much time as they wanted until they felt comfortable with their solution, although estimates of up to thirty minutes (example problem) and up to forty-five minutes (real problem) were verbally stated upfront. The instructions informed them that they could use the design technique in any way they saw fit, and that it was okay to ignore it altogether when appropriate.

7.5.2 Results

7.5.2.1 Resulting Designs

Subjects in both groups were able to produce reasonable designs. On reviewing subjects' work, it was apparent that five key design issues had been considered. The following list indicates these issues, along with the design questions which characterise each issue.

Laser Presentation How is the angle of each beam shown? How is beam thickness shown? How is the shape of the intersecting region represented? How are the two lasers distinguished from each other, and mapped to the parameter values?

Head Presentation and Monitoring How is the shape of the head presented? What is the representation of the orientation of the image with respect to the patient's face? How is the position of the 2-dimensional slice represented? How are critical regions of the brain indicated? How is head movement indicated? How is the radiologist alerted of head movement and how is excessive movement prevented?

Heart-Rate Monitoring How is heart-rate shown? Is the heart-rate trend represented? How is dangerous heart-rate prevented or alerted?

Starting, Stopping, and Quitting When the user tries to initiate firing, what assurances are in place that parameters are correct? What is shown while the lasers are operating? How easily can the user stop the operation, the restart it if desired? How does the user quit the system?

Parameter Entry How are the parameters and their units of measurement displayed? How is accuracy of parameters checked? Does the sofware assist with setting of some parameters?

The approach to each of these issues taken by each design team is shown in Appendix E. This section summarises and interprets the findings for each issue. Section 7.5.3.1 summarises observations pertaining to these issues as well as debriefing interviews. During analysis of results of pattern teams, situations where subjects had identified they had used a particular pattern were recorded.

7.5.2.1.1 Laser Presentation All teams represented an overhead view of lasers passing through the head. This is not surprising, as it resembles the presentation on the provided specification.

Pattern teams noted that an overhead representation is a demonstration of `Reality Mapping`. However, the pattern was of limited value given that it was already known to subjects. One team (Patt-B) recognised the utility of the `Interrogation` pattern, suggesting that users could zoom in on particular regions. Another team (Patt-A) applied `Redundant Information` to show figures for beam angle on the actual diagram (in its absence, the radiologist would either need to use a visual estimate or look at the parameters below). All three patterns are in the Information group, the most appropriate group for this issue.

One feature suggested by a principle team (Princ-A) but no pattern pair was to represent beam thickness by the thickness of the beam lines on the display.

7.5.2.1.2 Head Presentation and Monitoring Most groups showed critical regions (i.e. regions which should not be penetrated) and about half provided warnings or interruptions if the beams intersected these regions. Head orientation and location of 2-dimensional slice was represented by some teams. In general pattern teams correctly recognised the relevance of Machine Control patterns, i.e. `Warning`, `Redundant Information`, `Interlock`, `Shutdown`. In some cases, these patterns were used generatively; in other cases, they were simply used to label ideas which had already been applied.

An interesting generative application was observed in the Patt-D team. They were stepping through each pattern towards the end of the session and noticed `Recover`. This led them to think that the patient's head may be able to be re-aligned if any movement was detected. This, in turn, led them to propose a dialog box informing the radiologist of any head movement, prompting them to decide whether to continue. In this case, the pattern was identified as `Task Conjunction`, which, though slightly incorrect, is probably the closest pattern to the feature they had suggested. It would have been more accurate not to label it as any pattern. Nevertheless, the example demonstrates how consideration of one pattern can lead to an entirely different design feature.

Aside from the above example, there was little difference in the designs suggested by both groups.

7.5.2.1.3 Heart-Rate Monitoring In each group, three teams out of four proposed a graph of heart-rate across time. The pattern teams all identified this as an example of `Trend`. In debriefing, several teams from both groups indicated that the idea for a graph had come from the example case study they had encountered earlier, in which the example solution showed the sensor values across time. Thus, it is difficult to draw any

conclusions from the graph feature, other than to note that the pattern groups were at least able to correctly identify the pattern corresponding to their design.

The Patt-C team suggested an auditory warning for low heart-rates which sounds different to the auditory warning for head movement. The team obtained the idea in a generative manner by considering the `Distinct Interaction` pattern, which suggests tasks should be performed via different physical mechansims when there is a possibility of user confusion. The feature evolved into an example of the `Warning` pattern, which addresses distinct auditory warnings. The team identified the presence of the `Warning` pattern for the heart rate and head movement warnings, although they incorrectly labelled the distinct warnings as an example of `Distinct Interaction`.

7.5.2.1.4 Starting, Stopping, and Quitting Most subjects suggested a progress bar while firing, usually showing information such as time elapsed and remaining. A pronounced difference between teams was that three pattern teams proposed a confirmation box after radiologist hits Start button, whereas only one principle team (Princ-B) made this proposition. All three patterns teams were able to relate this feature to a design pattern. Two teams (Patt-A, Patt-D) correctly referred to it as a `Task Conjunction`, while the other team (Patt-B) called it a `Stepladder`. This stretches the definition of `Stepladder` since there is really only one intermediate step. That the patterns were related to a useful design feature is a good demonstration of the utility of the patterns; that the subjects did not converge on the same pattern indicates that the patterns may need to be stated more clearly, especially when there is a possibility of ambiguity.

The patterns team which did not provide a confirmation box did in fact provide a different check. Before the radiologist activates the beams, a simulation is shown with the intersection point highlighted. The radiologist cannot hit the Start button until the simulation was shown. While the simulation does not correspond to a pattern, the Start button restriction was correctly labelled a `Behaviour Constraint`.

The Princ-A team made several unique suggestions. First, the colour scheme of the entire view changes whenever firing is underway. Second, beam firing can be stopped by hitting any key and also by clicking on a large on-screen stop button. Third, quitting is only possible via a hidden (i.e. unknown to radiologist) key combination to avoid accidentally quitting and ensure only administrators can quit. All three suggestions have merit, and it is interesting to note that the patterns do not address any of them.

This is not to say the patterns *could not* address these features. For example, the first feature could be represented by a pattern suggesting designers indicate that a critical event is occurring, for the duration of the event[3]. Likewise, `Recover` could be reasonably extended to cover the second feature. To summarise, while the features are reasonable ideas, they are not represented in the present language. Had they been represented in the language, it is possible some of the patterns teams would have applied them. However, in their absence, the features may have been *more* difficult to think about, since the pattern groups were

[3]This would really be a pattern, rather than a generalisation of this particular instance, because other examples certainly exist; for instance, a red light showing while a tool is being activated, an audible tone when a truck is reversing.

engaged in applying the patterns. It is not possible to draw strong conclusions from this, since only one principle team produced these ideas. Nevertheless, this situation does serve as a reminder that there is a risk that application of patterns *might* cause designers to focus less attention on the domain and the fundamental context of the problem.

7.5.2.1.5 Parameter Entry All but one team (Princ-D) included some capability for checking parameters. The Patt-D team referred to this check as `Interrogation`, since it was manually invoked in their case. The Patt-A team had already suggested the feature, and experienced difficulty labelling it, identifying it as either `Behaviour Constraint` or `Automation`. This is another example where the study exposed some ambiguities among the patterns. This team also made a similar labelling error to the Patt-C team in the heart-rate issue; they proposed the beam width being shown as a bar with a tapered appearance to distinguish it from the intensity bar, and labelled this a `Distinct Interaction`. Technically, it is not a `Distinct Interaction` because it deals with presentation rather than user input. However, an information pattern such as `Distinct Presentation` could feasibly be inserted into the language, and it is understandable that the subjects did not notice the subtlety in the short time period they had to learn the patterns.

The Patt-A team suggested arrow widgets could be used to increment and decrement each time unit. As the team acknowledged, this idea was drawn directly from the syringe example in the provided case study. This is similar to the heart-rate graphs drawn by most subjects, which was based on the example case study. It suggests that the subjects are able to transfer knowledge in a relatively short time between an example and their own design problem. Interestingly, the Princ-A team suggested separating entry fields before and after the decimal point. This idea is similar in spirit to the syringe example, which the principle teams had not sighted. However, it is different in implementation. This further emphasises that the Patt-A team was generalising from the syringe example.

The Patt-B team generatively applied `Recover` to suggest an Undo button allowing parameter entry to be reversible.

7.5.2.2 Observations and De-briefing Interviews

This section documents the de-briefing interviews which were conducted for about ten minutes at the conclusion of the session, and also mentions observations made during the study. For the principles teams, only comments about the general technique are of interest. For the patterns teams, the following are discussed: pattern components which were used, whether relationships among patterns were used, and comments on the presentation and format of patterns.

7.5.2.2.1 Application of Design Technique (Table 7.1) In general, teams from both groups applied their own commonsense before applying their assigned design technique. However, there was an important

difference: the principles teams took little input from the principles, while the patterns teams tended to start off with commonsense, but then iterated between patterns and commonsense.

The principle teams essentially constructed their entire design by commonsense, and only referred to principles as a way of identifying what they were doing. They gradually focused less attention on the principles and developed according to their own ideas, returning to the principles as a checklist at the end. This was confirmed in their comments, which indicated that the principles were a checklist, and also suggested that the principles gave them a general sense of the type of design which was desired.

The example design problem, which both groups attempted, proved particularly beneficial to the patterns teams. They attempted to use the patterns to the example problem, and gradually became more familiar. Patt-B indicated that their initial unfamiliarity waned throughout the session.

The Patt-A and Patt-D teams began approaching the radiology problem with commonsense, but spent more time iterating between the patterns and their own ideas. They gradually became familiar with some of the pattern names and ideas, and attempted to incorporate them into their design *during* the design process. All pattern teams used the patterns as a a checklist, stepping through the patterns at the end to see if there was any applicability. The Patt-D team were unique in their references to patterns as a form of rationale, stating for example, that the approach "gives assurance that what you're doing is correct or on the path to being correct". The Patt-C team worked similarly to the principles teams, using patterns only near the end as a checklist. They were able to label several features as patterns as Section 7.5.2.1 shows, but did not generate anything significant from the patterns.

The Patt-B team provided an interesting insight into the organisation of patterns. They appeared very enthusiastic about the approach, but found the language structure did not support their design task. The language was structured according to relationships among the patterns, but they had particular design problems to solve, and wanted to know which patterns applied at a particular time. Thus, they spent about ten minutes re-categorising the patterns. They created four categories, based on a situation which has been identified, e.g. when hazardous states are identified, apply `Recover`, `Separation`, `Behaviour Constraint`. The other categories were Complex Tasks (`Stepladder`, `Separation`), Right Intentions (`Task Conjunction`, `Affordance`, `Distinct Interaction`), Recoverability (`Transaction`). This action was somewhat surprising, since the patterns are already arranged in groups according to type of design task (i.e. designing presentation; designing task flow, etc.); but the subjects were interested in which patterns would apply when certain situations are identified. It is difficult to draw too much from the team's categorisation, since it was prepared quickly and with little previous exposure to the patterns. However, it is important to note that the analysis in Section 7.5.2.1 suggests that the team did use the patterns relatively effectively, and the team also stated that the patterns were "useful tools for design". This experience raised the question of how a pattern language should be represented in order to be applicable to designers.

Table 7.1. Results from Safety-Usability Patterns study: Use of design techniques, mentioned by subjects in de-briefing interviews

Team	Observations
Princ-A	Mainly used commonsense. Also looked a little at principles for prevention and reduction rather than error recovery. Principles useful as a checklist but mostly looked over it briefly rather than as a formal checking mechanism.
Princ-B	Mainly used commonsense, but principles useful as a reminder and a check after the original idea. Some principles were difficult to understand — need further explanations or examples to show "what does that principle really mean?".
Princ-C	Mainly used commonsense, and principles as an afterthought. Principles important to "confirm choices".
Princ-D	Mainly used commonsense, with principles as a check at the end.
Patt-A	Started out using commonsense, then used patterns...led to iteration between commonsense and patterns. "Because we were using them as a checklist, they were constructively used"..."Certainly cases where just didn't [apply]. But there were always other areas that did in that case".
Patt-B	Initially experienced unfamiliarity. But eventually saw patterns as "useful tools for design", "a realisation of commonsense intuition"..."We clarified [i.e. categorised] patterns according to problem statements" because groupings in the paper not the best way to define it. Did not use principles much; they were "just generalisations of patterns".
Patt-C	Began with the presented problem and applied commonsense. Principles were hard to use and could be better phrased, e.g. Reduce Slips — "hard to see what they're getting at". Patterns "may be good to look at only after [applying commonsense]".
Patt-D	Used commonsense and principles. The patterns "verified" and "extended" this. Were "influenced by patterns when thinking about [design]". Also "gave us boxes to categorise when we're playing with ideas". "Gives assurance that what you're doing is correct or on the path to being correct". Easier to use patterns as a checklist; "since I haven't spent much time looking at these...easier to check afterwards [than determine which pattern is appropriate]".

Table 7.2. Results from Safety-Usability Patterns study: Components of patterns used by subjects, as mentioned during de-briefing interview

Team	Observations
Patt-A	Mostly used Context, Problem, Solution, Examples, Summaries, Case Studies.
Patt-B	Mainly used Summaries, Case Studies.
Patt-C	Mainly used Examples field, Case Studies.
Patt-D	Mainly used Forces and Resulting Context, Summaries, Case Studies.

7.5.2.2.2 Pattern Components Used (Table 7.2) The patterns teams were varied in their attention to the different aspects of the patterns. When asked which pattern fields they used, subjects were mixed across the major fields; in fact, Patt-A stated they used an inverse set of fields from the fields Patt-D had mentioned.

One particularly strong observation was that all teams were interested in the case studies. The subjects often alternated between the patterns and the case studies they referenced. The patterns are written assuming some knowledge of the case studies, so an ideal application would expose the subjects to the case study first. However, subjects were keen to apply the patterns immediately; hence, they tended to flip between the two. Another reason may have been the inclusion of diagrams in the case study but not in the patterns; the diagrams appeared to be a quick way for subjects to grasp key concepts (Section 7.5.2.2.4).

The other important observation was that subjects spent most of their time looking at the pattern summaries. This also proved to be a time-saver, enabling subjects a quick view and comparison of all patterns.

7.5.2.2.3 Relationship among Patterns (Table 7.3) Subjects reported a lack of interest in the relationships among patterns. Although they devoted considerable attention to the pattern summaries, they chose not to consider the dependencies between patterns. Patt-A did suggest the relationships helped them in clarifying the individual patterns, and Patt-D indicated that the pattern map might be useful for this purpose. The Patt-B team were certainly interested in the notion of grouping patterns, but as stated in Section 7.5.2.2.1, they did not find the existing groupings adequate for their task.

This result is not surprising. In a single design session, the subjects were most interested in immediately applying the patterns. Also, subjects were allowed to use the patterns in any way, rather than being instructed to follow through the network of delegation. This result does not say much about the efficacy of the language structure. The main point here is the evidence that language relationships can clarify the meaning of individual patterns.

Table 7.3. Results from Safety-Usability Patterns study: Application of relationships among patterns, as mentioned by subjects during de-briefing interviews

Team	Observations
Patt-A	Did not use relationships directly, but found relationships and Resulting Context section useful for understanding the patterns themselves. For example, using `Recover` as part of `Stepladder`.
Patt-B	Found groupings were not the best way to categorise. Did not use pattern diagram much — found connections were not always obvious. Would like to see examples of inter-pattern relationships.
Patt-C	Did not consider relationship among patterns.
Patt-D	Did not consider diagram. Could not see how diagram could be useful, although suggested it might serve as a memory aid for recalling the patterns.

Table 7.4. Results from Safety-Usability Patterns study: Comments on presentation and format of patterns during de-briefing interviews

Team	Comments
Patt-A	Helpful that important parts were boldfaced.
Patt-B	Too long to comprehend a large proportion of it. Mainly used summary. Could improve presentation with diagrams, like those in case studies.
Patt-C	Quite long and mainly used summary.
Patt-D	Mainly used summary, as descriptions were too long for this task.

7.5.2.2.4 Presentation and Format of Patterns (Table 7.4) Subjects offered a few interesting insights about the presentation of patterns. The tendency to use the summary sections noted previously was explained by three teams as a result of the length of the patterns. Interestingly, the Patt-A team, which did spend more time looking at the main pattern text, made use of the key fields being boldfaced. This was apparently a way to utilise the patterns while ensuring they were not overwhelmed by detail.

It appeared that all groups were drawn to the case studies in part by the diagrams, since they referred to the diagrams while discussing their design process. The Patt-B team suggested that diagrams would help make pattern descriptions more approachable, noting that they helped the case study descriptions.

7.5.3 Summary and Implications

7.5.3.1 General Observations

The study helped to clarify several issues about how designers interact with the safety-usability patterns, some of which can reasonably be generalised to general use of patterns.

The patterns seemed to have some effect on the resulting designs. For instance, the use of `Interrogation` to suggest zooming on particular brain regions, and the use of `Redundant Information` to overlay laser angles (Section 7.5.2.1.1). Subjects also used patterns as a starting point for reasoning which led to a design feature *not* corresponding to that pattern, e.g. the Patt-B team's used `Recover` to lead them to suggest a dialog box prompting radiologists to decide whether to continue upon detection of head movement. However, it should be noted that there were only a few examples where pattern teams produced features that principle teams had not considered. It is difficult to conclude much about this, given the short period of time the students had to deal with the problem and the patterns.

While the patterns' contribution to design was low, subjects did show they could usually label patterns in their design. In some cases, though, there were also some incorrect labels, which is probably a consequence of the short learning period combined with possible lack of clarity in the pattern definitions.

Sometimes, the label was *nothing more* than a label, i.e. the subjects had retrospectively associated their design feature with a pattern rather than used the pattern generatively. This in itself is a useful capability, since it suggests the practicality of the "common vocabulary" strength of pattern languages. If designers can label their design features according to consistent conventions, they can work effectively with each other and more importantly, interdisciplinary work is facilitated. Using techniques for reuse to support interdisciplinary work is one of the broad goals of this research.

Some problems with the patterns were also discovered. Subjects found the pattern descriptions too lengthy for their purposes. This in itself makes the language less approachable, and may also have contributed to confusion about the meanings of certain patterns. One reaction to this would be to disregard it on the basis that the study was not a realistic simulation of real-life pattern usage. This is partly true, since a good pattern language should contain patterns which are rich enough to communicate an underlying set of values. However, designers do need practical tools which can be applied readily. A language can be immediately applicable even if it lends itself to long-term application (just as a well-designed software application should accommodate both novices and experts). Attention must therefore be paid to the needs of people who are learning to use patterns while they apply them.

Another drawback to patterns may be the tendency to focus *too much* attention on them, at the expense of commonsense and a careful consideration of the design context. This problem seemed more prevalent during the example problem, while subjects were struggling with the general concept of patterns. However, there were aspects of the design which some principles teams covered but were not mentioned by patterns teams, e.g. several features noted by Princ-A in Section 7.5.2.1.4.

The study also demonstrated that people apply patterns in different ways. Patt-B tried to use patterns in a generative manner; Patt-A and Patt-D typically began with commonsense but patterns made significant contributions to their final solutions; Patt-C used patterns mainly to label their solutions. A possible extension to this observation is that patterns suit some people's design styles more than others.

A study of this nature is necessarily somewhat imprecise. Nevertheless, the detailed observation format has allowed several important insights to be gained. These observations lead to considerations for developers of pattern languages. Key implications will now be discussed.

7.5.3.2 Implications For Authors of Pattern Languages

The study provides several guidelines which authors of pattern languages should consider.

Demonstrate with Good Examples The heavy use of the case studies which was observed leads to the conclusion that good examples are essential in conveying the idea covered by a pattern.

Provide Summary Views The subjects also relied heavily on the list of summaries, a thumbnail view of the language. The boldfacing of problems and solutions was also helpful to one team. There may also be other approaches which improve immediate accessibility; this is a subject for further research.

Provide Visual Illustrations Part of the attraction to the case studies was apparently due to their inclusion of diagrams. In the case of safety-usability patterns, it was convenient to have isolated descriptions of the case studies since several patterns used each study. In general, though, it would probably be useful for patterns themselves to contain images. These images would usually be screenshots of the feature being demonstrated. However, it still seems useful to use *any* image which relates to the pattern, since images seem to make the patterns more accessible and would probably help to provide some character to each pattern. If patterns had been complemented with images, it is possible subjects would not have seen them as lengthy bodies of text with little differentiation.

Eliminate Ambiguities While the text-based presentation of patterns may have led to some confusion for subjects, there also appear to be some genuine ambiguities in the language. `Task Conjunction` seems most appropriate for a confirmation box, but `Recover` is technically correct too. This type of ambiguity will make a pattern language difficult to learn and apply. Hence, authors should try to avoid such ambiguities, or at least explicitly note the possibility and explain how it is resolved.

Provide Instructions For Use The Patt-B team highlighted that the language structure did not necessarily allow the designers to determine which pattern to use at a given time. The context field is important for this, but may present problems at a high-level, especially for a large language. The situation may be resolved by a structure such as Tidwell's Common Ground patterns [218], which are grouped according to questions (e.g. "What is the basic shape of the content?"). However, this may not always be appropriate. A more hierarchical language like Planet makes the subject less of an issue,

since there is a natural flow through the patterns. A more general solution may be to illustrate the use of patterns via sequences, a technique which Gabriel and Goldman used effectively in their Jini Community language [87]. This approach may be likened to a usage scenario for the language, i.e. a it shows how a hypothetical designer moves through a sequence of patterns as the design progresses. A hypertext presentation may also assist, as different index pages could be prepared depending on the type of usage and the individual who is using the patterns.

Observe Designers Interacting with Patterns Finally, this study has suggested that there are great benefits to observing people use a pattern language. It has exposed some ambiguities in the language (e.g. Section 7.5.2.1.4) and also provided candidate patterns which had previously been unconsidered (e.g. Section 7.5.2.1.5). Watching designers use the language would benefit pattern authors, whether by controlled experiments or informal workplace observations.

7.5.3.3 Implications For Developers

The observations in this study have suggested several implications which developers should consider before they adopt a pattern language.

Allow Time for Training In the short period allocated for this study, subjects exhibited considerable improvement in their understanding of the patterns and the pattern concept. Ideally, a pattern language will support learners, but this will not always be possible. Be prepared to spend some time learning the patterns.

Pay Attention to Design Context Patterns can be helpful but must be applied with caution. Patterns may not cover important aspects of the design. They are also no substituate for a deep understanding of the domain or the user's characteristics and tasks (e.g. the patterns teams did overlooked certain issues in Section 7.5.2.1.4). Use patterns where they help, use principles and experience where they do not.

Adopt a Comfortable Approach to Patterns The study suggests there may be individual differences in the way people use patterns. Therefore, as experience with patterns grows, develop an appropriate personal/organisational style instead of feeling compelled to make the pattern language the central focus of the development process.

Look at Examples Examples were useful in their own right, and appeared to enhance understanding of the patterns. Therefore, enhance understanding of adopted patterns by determining how they relate to any systems which are available for analysis. Gabriel's [57] comments in Section 3.2.3.2 remind us that patterns are present even in systems that were not explicitly designed according to patterns. Classifying patterns in existing systems should improve understanding of the adopted patterns and may also pave the way for enhancements to the language.

7.6 Discussion

The safety-usability language is a further example of a usability pattern language constrained by specialised requirement. As with Planet, the constraint provides it with a greater capability to unify different HCI constructs. Where Planet deals mostly with organisational process, user-interface elements, and functionality, the safety-usability patterns bring together task flow, user-interface elements, input mechanisms, and machine intervention. The languages are also both based on explicitly stated design principles.

The safety-usability patterns have been studied via an empirical study, with conclusions discussed in Section 7.5.3. The study was beneficial at a number of levels. It helped to assess the safety-usability pattern language, in terms of the individual patterns, the overall structure and purpose, and the presentation. Several observations also seemed broadly applicable to pattern languages in general, and guidelines for authors and developers have been noted. These guidelines cannot be considered conclusive, but they do seem logical at face value. It would be useful to perform further study with patterns constructed according to such guidelines. Indeed, the study raised several questions for future research:

- How do individuals vary in their approach to patterns, and what guidelines for pattern usage follow from such variation?
- How can a language be represented, including any complementary material such as overview diagrams, to enable designers to easily pinpoint the pattern(s) which should be used at a particular time?
- The study was performed by two people with similar backgrounds, but it would be interesting to observe a pattern-centered interdisciplinary design session.
- Another limitation of the study was that it focused on a relatively small problem. It would be interesting to perform a longer-term, ethnographic, study of patterns in the workplace. This would not only enable insights into the interdisciplinary process, but also the evolution of a design in the context of continual external forces and ongoing modifications. There were no strong differences between designs of pattern teams and those of principle teams in the experiment performed here. A longer study would allow designers to become familiar with the patterns, and it would be possible to provide a more definitive conclusion about the impact of patterns on design.

The Planet language and its associated online repository concept, combined with the safety-usability language, have served to demonstrate what is meant by a high-level usability pattern language. The *language* aspect is demonstrated by the high degree of connectedness among the various patterns. More work is required to show how low-level pattern languages can also improve usability, and also to examine the relationship between high-level and low-level reuse. This line of research is the subject of the next part of the thesis.

Part III

Detailed Design Reuse

Chapter 8

Generic Tasks: Theory and Empirical Study

8.1 Introduction

In shifting focus from high-level patterns to low-level patterns, it is important to remember that one of the main goals of this project is to show how the varying levels of abstraction are related. The following chapters work towards MMVC, a framework containing software components and a pattern language for detailed software design. The second aim of this thesis is to consider how to reuse detailed design features that have contributed to usability, and MMVC demonstrates this possibility. It consists of a set of detailed design patterns that can aid a developer in constructing software which supports certain functionality, such as ”sort a list”. The functionality itself is determined by generic tasks, i.e. tasks which recur in many systems. Generic tasks, such as `Compare Two Documents` and `Undo a Command`, were chosen as a useful starting point because it was reasoned that their implementation would be similar in different software, so long as the software had the same underlying architecture.

The details of MMVC and the application of generic tasks to patterns are deferred until Chapter 10. This chapter focuses on the general concept of generic tasks, showing how a set of generic tasks may be derived and demonstrating their applicability for design. The concept is introduced in Section 8.2, and its relevance to the thesis is explained. The methodology used for obtaining a list of tasks is discussed in Section 8.3 and the task list itself shown in Section 8.4. Section 8.5 outlines a pilot study conducted with subjects using the generic tasks for brainstorming. Further applications for generic tasks are listed in Section 8.6. Section 8.7 is a discussion of the findings.

8.2 Generic Tasks: An Overview

A generic task is in some respects a simplified version of a task-based pattern, the type of pattern described in Section 4.2.1.1. An approach like Breedvelt-Schouten et al.'s [30] looks at recurring patterns in task models, e.g. a multi-value input consists of editing attribute 1, editing attribute 2, and so on. Sutcliffe and Carroll have described an ongoing project to classify what they also call "generic tasks" [212]. Like Breedvelt-Schouten et al.'s patterns, a broad task is identified and a sequence of actions corresponding to the task are documented. Their "diagnosis" generic task, for example, consists of several steps: determining the cause of malfunction, locating its cause, and proposing remedial treatment. These approaches are useful and can provide a rich basis for reuse of high-level HCI knowledge. Sutcliffe and Carroll, for example, have used their generic tasks as an input to reusable claims about HCI phenomena.

In the present work, however, the generic tasks are simply defined as unit actions such as "Edit" and "Specify Value", with less focus on the relationships between tasks. This was more appropriate given the way the generic tasks are used: for brainstorming and as an input to the MMVC framework. The generic tasks cannot be considered proper design patterns because the problem each generic task solves is not articulated, nor are the typical fields such as forces and examples.

Foley et al.'s [82] concept of "basic interaction tasks" for interactive graphics maps closely to the notion of generic tasks described here. For programs modelling two-dimensional space, the authors proposed four basic tasks: specifying a position, selecting an element, interacting with text, and specifying a numeric value. The taxonomy also describes three "composite interaction tasks": dialogue boxes, construction techniques (e.g. specifying a line), and dynamic manipulation (adjusting previously-constructed objects, e.g. rotating an object). They analyse each task, describing interaction mechanisms which support it and variables involved. Specifying a position has a resolution variable, for instance, and this will depend on whether input comes from keyboard, joystick, and so on.

More recently, Baber [9] described several "generic actions" of human-computer interaction (HCI), on the basis of ISO/DP 7942. Similar to Foley et al., the purpose was to establish a framework for discussing how various input devices compare in supporting particular tasks. Under this scheme, four categories were identified (with a total of seven actions): selecting objects, dragging objects, changing the orientation of objects (e.g. rotation), and entering data.

The work described here extends these contributions by identifying a broad list of generic tasks. The analysis has been performed in a systematic manner which could be repeated, for instance, by an organisation wishing to capture recurring tasks among its own projects. The purpose was to show how a set of tasks could be established from an existing corpus of software. Since the generic tasks are used to help generate the MMVC framework (Chapter 10), the work here demonstrates how an organisation could move from an existing set of applications to a reusable framework which supports the development of related software.

The envisioned process is represented in Figure 8.2. Generic tasks are used in conjunction with regular

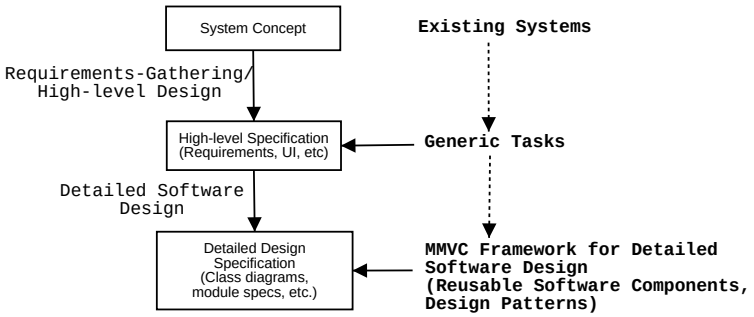


Figure 8.1. Envisioned development process for generic tasks and MMVC.

techniques to produce a high-level specification. The MMVC framework has been developed in accordance with a certain subset of these generic tasks. Therefore, MMVC will help detailed designers and programmers implement systems which support these particular tasks.

Sample programs are used to demonstrate how MMVC can provide detailed design assistance to developers implementing the generic tasks. This chapter is concerned with the process of gathering the tasks in the first place, and furthermore, using them in a brainstorming capacity. The latter issue is explored via the pilot experiment discussed in the latter sections of this chapter.

.

8.3 Identifying Generic Tasks

To derive a list of generic tasks, fourteen software projects were analysed. The projects were performed by teams of about four people, as a full-year third or fourth-year software project, each producing in the order of 10,000 lines of code. Clients were drawn from both industry and academia.

Fifty candidate projects were narrowed down to the final fourteen according to two criteria:

Human-computer interaction The system had to interact directly with humans.

Complete documentation Complete software requirement specifications and user documentation were necessary. Requirements specifications had to contain usage scenarios.

There were no criteria relating to the the final product. In fact, the final product's usability was not considered because the focus was *tasks* which emerged after requirements-gathering, rather than the ways in which the software supported these tasks.

Key tasks were extracted from the usage scenarios. This allowed for quick identification of a large number of tasks from a variety of domains. However, there was a concern about the depth of this approach; it is conceivable that particular families of tasks are systematically left out of these scenarios. Furthermore,

Table 8.1. Descriptions and sample generic tasks for projects which underwent task analyses.

Project Description	Key Tasks
Contractor selection tool: To produce a "shortlist" of tenderers	Summarise, Add, Remove (tenderers); Navigate (through quality factors)
Timetabling tool: To construct tutorial timetables	Change Display Options (of timetable view); Add, Remove, Move (details); Update System State (to check against domain constraints);
3D Weather Visualiser: To explore weather prediction data	Change Display Options (e.g. wire-frame vs. solid), Enter data (display option parameters).
Radar Tracking System: To visualise objects being tracked by a radar	Navigate (to different map areas), Obtain guidance (about how to use the system).
Source Code Visualiser: To help programmers visualise source code	Change Display Options (e.g. direction of arrows), Add, Remove, Move (nodes representing source code modules).
Multi-lingual Botany Database: To enable queries to a plant database	Search (for matching data), Change Task Parameters (to set query language).
Online Tutorial System: To support real-time online interaction between tutors and students	Navigate (tutorial material), Enter Data (to answer questions).

scenarios are only single episodes and can be difficult to generalise from — for example, to reason about common task sequences.

Seven diverse projects were selected for detailed task analysis (Table 8.1), although key tasks from the other seven were also retained. The analysis looked in more detail at the requirements specification, thus tasks could be considered even if they didn't feature in the documented usage scenarios or user documentation. There were three principles in selecting the projects to undergo task analyses: (a) a diverse cross-section of projects was required; (b) relatively complex requirements were favoured; (c) successful projects were preferred. User documentation and requirements specifications were consulted to extract task models for the systems. The JSD* notation (see Section 2.3.3) was used to document the models (Figure 8.3). JSD* was appropriate in this situation primarily because it is graphical and has a well-defined syntax, although it was by no means an essential choice. In combination, these attributes facilitate the process of analysing typical patterns.

8.4 Resultant List of Generic Tasks

After recording tasks for each project and placing them into categories, the classification of Table 8.2 emerged. Although exploring these relationships is beyond the scope of this study, it is worthwhile acknowledging a few ways the tasks can be related to each other. One possible relationship is for a task to be a special case of another task. Another relationship is where two tasks complement each other. `Search` and

Table 8.2. List of generic tasks, ordered by category. The "Example" column shows instances of the tasks captured from the projects being analysed. The "Word-Processor" column demonstrates the applicability of the generic tasks to a typical application—Microsoft Word 97 [60].

Tasks	Example from Analysed Projects	Corresponding Word-Processor Task
Browse Material		
Search	Search for text from an online product database.	Search for text pattern within document.
Navigate	Request information about an element in a multimedia chemistry tutorial.	Scroll to a desired text region.
Summarise	Show a summary of objects which have been shown in an animation.	Perform word count.
Compare	Compare a weather chart on two different days.	Track document changes by comparing two different versions.
Change Display Options	Select desired language of query results.	Enable text-wrapping.
Specify Material		
Enter Data	Enter data from keyboard.	Enter data from keyboard.
Select	Choose an internet relay channel.	Select a font.
Compose	Group together several graph nodes to form a single node representing the entire group.	Select several drawing objects at once.
Decompose	Ungroup a grouped node.	Select an individual object when several drawing objects have been selected.
Change Material		
Add	Add a new lesson to a multimedia tutorial.	Add text.
Remove	Delete a company from a list of tenderers.	Delete text.
Replace	Replace currently-loaded video sequence with a new sequence	Over-write a selected text block.
Move	In a timetable representation, drag a subject cell to a new timeslot.	Move a drawing object.
Transform	Re-layout a network of nodes and edges.	Underline a text block.
Handle Entire Material		
Store	Save a recently-generated animation	Cut text to clipboard.
Retrieve from Storage	Load source code which is to be visualised.	Paste from clipboard.
Duplicate	Copy a file	Copy to clipboard.
Meta-Tasks		
Preview Task Effect	Change cursor to indicate effect of clicking within a certain area of a multimedia tutorial	View a preview of the document as it will appear after printing.
Obtain Guidance	Request context-sensitive help by clicking on a Help link	Select Help from an object's Properties menu.
Undo	Click Back button to return to previous page.	Reverse a text change.
Update System State	Click Verify button to see timetable cells which meet specified constraints.	Update an Include Text Field after a change to the file corresponding to this field.
Change Task Parameters	On a graph visualiser, modify default number of expansion levels which apply when a user adds a node's ancestors or descendants.	Change text entry mode from Insertion to Overtype.

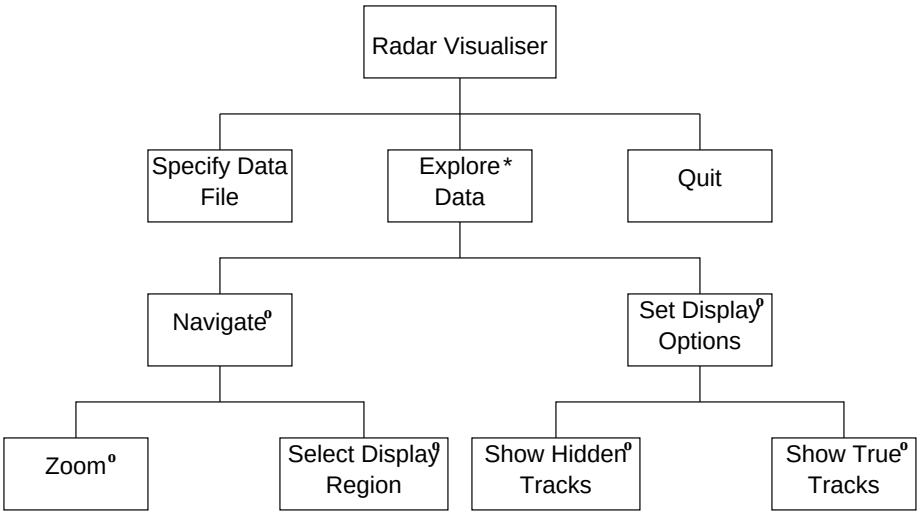


Figure 8.2. A sample hierarchical task model in JSD* notation demonstrating tasks for a radar tracking visualisation tool. Left-to-right boxes indicate sequence (i.e. "Explore" follows "Start"). "*" indicates iteration (i.e. the user can perform any number of "Explore" tasks). "°" indicates selection (i.e. An "Explore" task can either be "Navigate" or "Set Display Options").

`Navigate` could be used in tandem by a user trying to locate a passage of text, for example. Sometimes two tasks are necessarily connected to each other, but can be performed in either order, e.g. when the system allows either "noun-verb" or "verb-noun" actions.

The tasks were placed into five groups. Note that the term *material* is used to describe software artifacts, after Riehle and Züllighoven ([194]; described in Section 4.3.10). The groups are:

Browse Material These tasks help a user explore a material, and might be used for viewing only, or to ultimately changing of material.

Specify Material These tasks help users specify information to the computer, such as choosing from a list of items. The aim is to perform some task with the selected material, such as changing it.

Change Material These tasks enable the user to add, remove, or change the material.

Handle Entire Material These tasks do not change a material, but operate on it as a whole. For instance, saving a file is an operation on a material which does not actually change its contents or structure.

Meta-tasks Meta-tasks moderate or expose the effects of other tasks. They would not be very useful in isolation, but they can make tremendous contributions to overall usability when other tasks are possible. For instance, there are often functions which allow users to change the way certain tasks work, enabling a flexible workflow.

The second column of the table shows examples of the tasks extracted from the selected projects. The third column demonstrates the external validity of these tasks by showing how each is supported by a typical word-processor, Microsoft Word 97 [60] in this case. The list presents a wide range of tasks, but no attempt has been made to produce an exhaustive collection of recurring tasks. The purpose is to show that generic tasks can be systematically captured, and to investigate how such a list might be used.

8.5 Generic Task Application: Pilot Study Description

The application of generic tasks considered in this chapter is to assist with rapid generation of ideas for new software functions. A pilot experiment was conducted to investigate whether such a brainstorming technique might be useful. Furthermore, the generalisability of tasks could be demonstrated by taking generic tasks from one set of systems, and having subjects apply them to other systems.

8.5.1 Method

The methodology was to provide subjects with a standard structured method for devising new tasks, and then investigate whether introducing the generic task list could help them think of additional tasks. Six subjects volunteered to participate in the pilot study, all students at the Department of Computer Science and Software Engineering at the University of Melbourne. Four were postgraduates and two were final-year undergraduates. All students had extensive programming experience. The subjects sat in individual sessions lasting 1.5-2 hours. Materials provided to subjects may be found in Appendix F.

Two hypothetical systems were initially shown to subjects. One was ShopCart, a client-side application for internet shopping (independent of a web browser), and the other was Ready Reader, a hand-held reading device (Figure 8.3). Several sketches of screens were provided for each, along with accompanying descriptions of possible tasks at each stage. Written instructions informed the subjects about the techniques they were being asked to employ.

In the first phase, subjects could use any means to produce the tasks. To make this stage comparable to the second phase involving generic tasks, a structured method was offered; however, it was made clear that this was only a brainstorming tool, and any other ideas were encouraged. The structured method involved asking subjects to write down users' goals in using the system, and then to identify the problems users might have in achieving these goals. From these goals and problems, subjects were asked to write additional tasks which users might be able to perform with the system. Subjects were given guidance on the structured technique, as well as a worked example, and were informed that it did not matter if they got the concepts confused (e.g. defined something as a task instead of as a goal) because the administrator would interpret their work later on. The aim here was to encourage subjects to provide as much information as possible before being introduced to generic tasks.

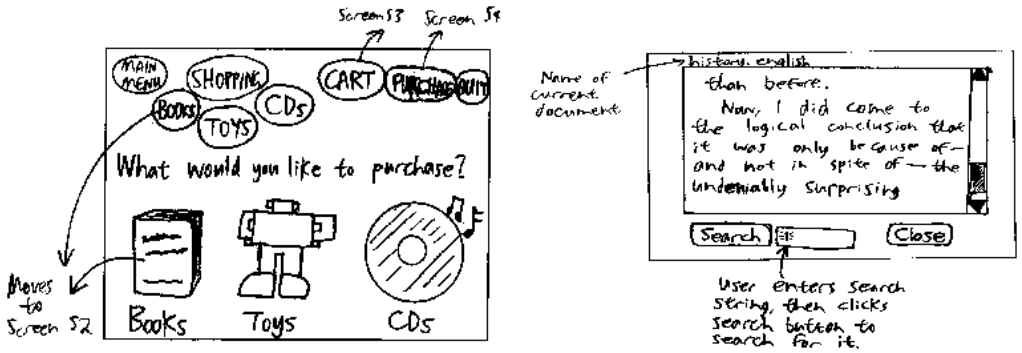


Figure 8.3. Material for generic task study: Sample hand-drawn sketches for ShopCart and Ready Reader applications. ShopCart screen shown is main menu, one of five screens. Ready Reader screen is document view, one of two screens.

When subjects felt they had exhausted their ideas for each system, they entered the second phase. The generic task list was introduced to them as an alternative method, from which more tasks could be added to their initial list. Subjects were asked to use the generic task list in a similar manner to the first approach, i.e. as a flexible brainstorming tool, so that they could feel free to suggest tasks which may not correspond directly to the generic tasks. To provide some structure, subjects were asked to step through each task, recording ideas for the system as they proceeded. Then, they were asked to write any other tasks they could think of. Finally, they were asked to step through the list and check if anything could be added. The experiment concluded with a semi-structured de-briefing discussion to clarify subjects' work and compare the approaches.

8.5.2 Results and Discussion of Study

Subjects' work was initially processed, with any identified goals or problems being converted to tasks if that was more appropriate. Tasks were eliminated if they seemed more like goals or problems, and counted once only if the person repeated them (it was quite common for people to rewrite a Phase 1 task as a generic task).

Many tasks proposed before the generic task list was introduced nevertheless mapped to a particular task (e.g. many people suggested a search function in the initial phase). Conversely, some tasks proposed after the list was introduced did not map to any particular generic task. Thus, tasks were classified according to (a) which phase they were identified in, and (b) whether or not they fitted a generic task.

8.5.2.1 Examples of new tasks

To illustrate the type of outputs subjects produced, I now describe the tasks which were frequently proposed in each phase. Following is a list of ShopCart tasks proposed by three or more people during Phase 1,

accompanied by the matching generic task (the figures in parentheses indicate how many subjects listed each task).

Phase 1:	
Search for item (6)	Search
Remove item from cart (4)	Remove
Look at previous carts (4)	Retrieve
Phase 2:	
Specify item quantity (5)	Duplicate
Change language (3)	Change Display Options
Help on usage (3)	Obtain Guidance

For Ready Reader, tasks suggested by three or more people in the Ready Reader were as follows:

Phase 1:	
Show tree/list of all files at once (4)	Summarise
Search for word across documents (4)	Search
Create a new folder (3)	Add
Search for documents, e.g. by date (3)	Search
Phase 2:	
Change font (3)	Change Display Options
Group documents for delete/move (3)	Compose

It is also important to consider specific useful tasks which arose from the generic task concept. Some examples of tasks which were *only* suggested during Phase 2 of the ShopCart application are shown below. It is interesting to note that most of these tasks are supported by existing e-commerce sites, but the generic task list seemed to trigger subjects to consider them.

- Move to “I might buy later” area (Move).
- Compare current cart with a previous cart (Compare).
- Compare among products in the same category (Compare).
- Change currency (Change Display Options).
- Select several items at once, e.g. for deleting or puchasing (Compose).
- Choose between deliver all products at once or delivering as they are available (Change Task Parameters).

Useful or novel tasks proposed only in Phase 2 for Ready Reader were:

- Change document orientation (Change Display Options).
- Show space used or free (Summarise).
- Show language of document or change language (Change Display Options).
- Merge documents (Compose).
- Summarise bookmarks (Summarise, after the subject suggested allowing bookmarks).

That subjects considered these possibilities only after being exposed to the generic tasks lends support to the notion that brainstorming can improve quality and reduce the likelihood of tasks being overlooked.

8.5.2.2 Comparison of methods

Table 8.3. Results from generic task study: Number of tasks generated before and after generic task list, for each subject. For each subject, number of tasks generated before generic task list shown (Phase 1) and after generic task list shown (Phase 2). Mean and standard deviation are also shown.

	ShopCart			Ready Reader		
Subject #	**Phase 1**	**New in Phase 2**	**Total**	**Phase 1**	**New in Phase 2**	**Total**
A	8	2	10	6	5	11
B	13	7	20	2	24	26
C	13	6	19	2	6	8
D	12	12	24	7	14	21
E	9	6	15	2	8	10
F	4	1	5	8	3	11
Average No. of Tasks	9.8	5.7	15.5	4.5	10.0	14.5

Overall, both phases yielded results in favour of the generic task intervention, as shown in Table 8.3. Mean values were calculated, but no statistical test performed. It is clear, however, that the generic task list had a marked effect. All subjects were able to think of several tasks before being shown the generic task list, and introducing the list led them to add several more. This result was true for both hypothetical systems.

One interesting result is the wide individual variation which occurred for both phases. This is not entirely surprising, since people will have different abilities in solving this type of problem. Some people seem to gain a lot of support using the generic tasks (Subjects B and D), while others appear to derive less benefit (Subject A). All subjects experienced relative increases in usage of generic tasks during the second exercise, Ready Reader. De-briefing discussions suggested that this was due to a learning effect in which the unusual concept of generic tasks becomes more familiar.

8.5.2.3 Number of unique tasks generated

The previous table considered individual performances, so that a task was counted twice if two people suggested it. In this section, tasks are considered as being generated by the group as a whole. In other words, the description considers number of unique tasks generated.

Table 8.4 shows that in both applications, introducing the generic tasks more than doubled the number of unique tasks suggested. This is despite the fact that individuals added fewer tasks on average in Phase 2 than they initially proposed in Phase 1 (Table 8.3). This demonstrates that generic tasks led to more variation among individuals; they led to many tasks which only one individual thought of. In contrast, there was more commonality among the initial tasks. People commonly proposed typical tasks such as searching for a book and showing total price for the cart.

The table also shows that quite a few Phase 1 tasks mapped to a generic task. This makes sense,

Table 8.4. Unique tasks generated by the overall group. The top row shows tasks which mapped closely to generic tasks (e.g. "Group items" relates to compose) and the second row shows tasks which does not relate strongly to any of the generic tasks. "Only in Phase 1" shows tasks generated before the generic task list was introduced (even though many suggestions mapped to generic tasks), "Only in Phase 2" shows those after the list was introduced, and "In Both Phases" shows tasks which at least one person suggested before and at least one person suggested afterwards.

	ShopCart				Ready Reader			
Task Type	**Only in Phase 1**	**Only in Phase 2**	**In Both Phases**	**Total**	**Only in Phase 1**	**Only in Phase 2**	**In Both Phases**	**Total**
Generic mapping	9	30	7	46	14	22	9	45
No generic mapping	12	1	0	13	6	2	0	8
Total	21	31	7	59	20	24	9	53

as designers presumably have their own models of generic tasks, some of which corresponds to those considered in this investigation. In de-briefing conversations, subjects indicated that in the first phase they were often considering software they had seen before. Several people commented that they had features of Amazon.com in mind when designing ShopCart.

Many tasks were suggested in Phase 2 only. A concern about this methodology is that subjects may not consider items important enough to record in the first phase, but will record them in the second phase if they see a corresponding generic task. This may have happened in a couple of instances. For instance, a Help function was not recorded initially, but on seeing the generic task "Obtain Guidance", three subjects proposed it. However, this was not the case for most tasks. Two useful Ready Reader tasks suggested by three people in Phase 2 but overlooked by everyone during Phase 1 were (a) grouping documents together for deleting or moving ("Compose") and (b) changing font ("Change Display Options"). Furthermore, the "Both Phases" column suggests that at least for some tasks, there can be high assurance that the generic list was helpful. This column shows that there were some tasks which some users thought of initially, and others thought of later on.

8.5.2.4 De-briefing discussion

Most subjects reported some difficulties using the first technique to map from goals to problems to tasks. They had difficulties understanding the distinction between goals and tasks, and also found there was a very close mapping between problems and tasks. For instance, a typical problem was "Does not have Search", and the corresponding task was simply "Let user Search". However, subjects were asked to show mappings between goals, problems, and tasks. This helped to check how close the mappings were. It turned out that subjects were not overly-constrained by this technique, since they chose to add many tasks which did not correspond to particular goals or problems.

All subjects stated that they found it relatively easy to map from generic tasks to specific tasks in the

system. Subject F — who did not perform many mappings — stated that he did not have problems doing this, but felt that he had proposed the major tasks already.

One surprise was that several subjects perceived the two phases as a cohesive methodology. I had intended the first phase as an initial benchmark for the generic tasks, and structured the instructions to give the impression that two separate techniques were being investigated (subjects were not informed whether any of the techniques was created by the researchers). In fact, one Subject (Subject D) suggested that the goals helped to consider tasks based on users' perspectives, and the generic tasks act as a safeguard to ensure that any other useful tasks had not been omitted. Another (Subject F) confirmed this view, stating that if she had started with generic tasks she may not have covered everything. Therefore, she explained, it was good to start by considering users' goals, then moving on to tasks.

8.6 Further Applications of Generic Tasks

The pilot study demonstrated one useful application of generic tasks, i.e. to improve brainstorming of user activity. This section highlights the variety of applications generic tasks could facilitate.

8.6.1 Supporting design reuse

Supporting design reuse is the major interest in generic tasks in this thesis. The generic tasks themselves are not design patterns because unlike patterns, they do not encapsulate an insightful solution to a difficult problem. However, they can act as a useful input to the pattern identification process, e.g. Breedvelt-Schouten et al.'s work shows how patterns of hierarchical task models may be captured [30]. Since software design should naturally flow from an understanding of tasks, it is feasible that authors of detailed software design patterns could also gain from an appreciation of which tasks are commonly required by users. This is the idea behind MMVC in Chapter 10.

8.6.2 Common Vocabulary for the HCI Community

Generic tasks are a useful frame of reference for expressing the rationale behind designs. For example, one can observe that deleting a file is performed by dragging to the trashcan on the Apple, and typing `del [filename]` in MS-DOS, implying a general "Remove" task. This enables a comparison of the strengths and weaknesses for users of different characteristics. It is also possible to look at the "Remove" task in a more general sense. Such a perspective would lead one to consider how a block of text is deleted in a word-processor, a user is removed from a network, or even a station removed from a television's memory of clearly-broadcast channels.

It is therefore possible to characterise and compare application styles, or specific applications, based on the way generic tasks are supported. This would be analogous to the standard properties computer scientists

ascribe to similar algorithms with a view to determining the circumstances where it is most appropriate to use one or the other, and the ways in which they might be usefully combined. One technique for sorting numbers might be considered appropriate for completely random data, whereas another technique might be more helpful if the list is partially sorted. In the same way, it might be said that a particular design would be useful for people who wish to "Compare" many items, but less helpful if the primary tasks are "Add", "Remove", and "Move". As with Foley et al.'s [82] and Baber's [9] "generic actions" described earlier, it is also possible to compare how different interaction devices enable users to perform generic tasks. Support for different kinds of users could also be compared in this way.

8.6.3 Interdisciplinary communication

The common vocabulary can extend beyond HCI boundaries; like patterns, generic tasks can have positive implications for interdisciplinary communication. Being able to specify requirements in terms of generic tasks would make communication simpler for everyone. A marketing specialist could structure a research survey around potential tasks to be provided, and forward recommendations in this format. Human factors experts, aware of the ways similar tasks have been supported in previous systems, could reuse existing designs to support these tasks. By explicitly informing software engineers of the tasks they are supporting, they can more easily adapt previous code, or previous ideas.

8.6.4 Novel design

One finding from the experiment was that generic tasks helped subjects think of novel tasks, such as a summary of bookmarks. Many designs which we consider to be "novel" still retain many features from prior art. Paradigms like new window managers and palmtop devices do change the way users interact and certainly pave the way for new tasks. However, many of those tasks closely resemble certain tasks in command-line interfaces and possibly even before computers were invented (e.g. it is possible to compare two documents without a computer). Generic tasks have the capacity to be applied to new devices. For instance, a designer of a DVD player might decide to "Summarise" the DVD by showing every 1000th frame. Combined with a user-centered investigation, this kind of application would help practitioners avoid redesigning the wheel every time a new system is called for.

8.7 Discussion

The first half of this chapter explained how studying tasks across a wide range of systems led to a generic task list. This process demonstrated that certain tasks do recur across different programs and that generic tasks can be systematically documented. The initial classification may prove of some use to developers, as well as the method for capturing generic tasks.

In the second half, a pilot study was discussed which demonstrated that generic tasks can have practical implications for designers. A repeated measures methodology was adopted, in which the generic task list was introduced once subjects felt their ideas had been exhausted. Since there was no control group, it is impossible to be certain that the subjects would not have arrived at the new tasks without the generic tasks being provided. However, there is significant evidence to suggest the generic tasks themselves had at least a significant influence on the ideas of subjects during the second phase. Firstly, the second phase took place immediately after the first phase, rather than a day or even an hour later. Secondly, many of the tasks proposed during the second phase were related to the generic tasks, with subjects indicating the associations. Thirdly, the de-briefing discussions indicated that subjects felt comfortable with the generic task concept and most felt they had been able to apply them directly to the design problem.

A useful property of this brainstorming technique is that it only needs one or two hours of work, even in an industrial setting (although several iterations may be required). Thus, the experiment was similar in duration to that of an industry setting. The experiment did not use contrived, small-scale, systems as examples; the hypothetical ShopCart and Ready Reader applications have a similar degree of complexity to many real-life projects.

It would be useful to expand the technique into a more integrative process which takes into account users' goals and contexts. The technique might also involve participative design, since it should be very easy for users to comprehend the generic task concept. Further research might form more quantitative findings related to factors such as individual differences and application type. Furthermore, it has been established that subjects were able to generate new ideas, but what is the quality of the ideas? It might be possible to perform further analysis of the data to this effect, perhaps by asking candidate users to rank desired functionality (using a list of tasks from both phases) [210].

The brainstorming technique can be used to design for systems which have no obvious direct analogy. Researchers have previously suggested the consideration of related systems, e.g. MUSE's extant system analysis [135], and the competitive analysis described in Nielsen's Usability Engineering approach [172]. Generic tasks provide developers with a knowledge base, leading to greater confidence that existing systems have been considered.

The notion of generic tasks may be most beneficial when applied to a particular organisation's projects or confined to a specific domain. Using a method similar to the one presented here, it would be possible to "mine" tasks typically supported by a certain class of software. Databases, for example, support tasks such as "View record", "Filter records", "Search for data". Such a tool might prevent late change requests for a developer working on small projects where large-scale usability studies are prohibitive; for example, someone developing a database for video library holdings or staff management.

The experiment focused on one application of generic tasks, i.e. as a rapid brainstorming technique. In fact, there is a broad spectrum of possible applications for generic tasks (Section 8.6). Reusable HCI knowledge bases and design techniques can be informed by an understanding of the tasks users typically

perform. The interdisciplinary HCI community can benefit from a common vocabulary; when a human factors specialist asks a software engineer to implement a “Change Display Options” task, the software engineer has a pool of knowledge to draw upon about how generic tasks may be implemented in different environments. In this way, generic tasks raise the level of abstraction from commonly-known GUI elements (“scrollbar”, “button”) to user tasks. Researchers can also benefit, by being able to compare interaction styles according to baseline generic tasks.

In this thesis, the main applications considered for generic tasks are those concerning their application in software engineering projects. The work here has established that a list of generic tasks can be obtained from a number of applications, and that such a list can be used to facilitate brainstorming with a relatively small investment of time. This supports the goal of using generic tasks in a higher level concept. In Chapter 10, the next stage of development is explored, i.e. the MMVC framework for detailed design which helps developers implement a set of generic tasks. Before this can be done, it is necessary to study the Model-View-Controller (MVC) framework on which MMVC is based. Chapter 9 introduces MVC and explores the issues involved in producing an MVC application.

Chapter 9

Refining the Model-View Controller (MVC) Architectural Pattern

9.1 Introduction

The aim of MMVC is to expose patterns which arise when creating software which supports generic tasks. Thus, MMVC takes a subset of generic tasks covered in the previous chapter, and shows the design structures which support those tasks. But there are many different styles of architectural patterns (also known as "architectural styles") in software. In general, there are patterns such as `Pipes and Filters`, `Blackboard`, and `Broker` [34]. Even in the specific area of interactive systems, a multitude of architectural patterns exist (see Coutaz's overview [62]). These patterns guide the overall composition of system architectures, and it would be possible to implement the generic tasks in any one of them.

A guiding principle in this thesis is that the scope of work should be constrained in order to maximise the coherence among resulting patterns. In Planet and Safety-Usability Patterns, the constraints were based on specialised requirements. In the present context, a constrained scope means selecting one particular architectural style. In this way, a system could be built under this architectural style which supports some generic tasks. One would expect deeply-related low-level patterns to emerge from systems which implemented the same tasks under the same architectural framework.

After deciding to focus on one particular style, the obvious question which arose: which style? It was logical to select an architectural style based on user-interface, but even this decision leaves open a wide range of choices. There are older, more straightforward styles such as Seeheim, Model-View-Controller, Presentation-Abstraction-Control, as well as sophisticated modern styles [62].

For proof-of-concept, a simpler style was sufficient. Following this choice, MVC was selected primarily because, although it is relatively old, its concepts have been influential in the construction of modern user-interface libraries. Hence, it was felt there would be more support for work within this architectural

style.

As work proceeded on an initial prototype, it became apparent that MVC was not as straightforward as it had promised to be. It was frustrating to learn that the literature rarely discusses the approach to designs with more than one model. And there have been few attempts to document multiple-model systems. Those applications which do contain more than one model usually deviate significantly from MVC conventions (e.g. [99, 117]). This may be with good justification, but it does not help to clarify how a multiple-model design can be implemented in MVC. Thus, it was decided to develop a multiple-model prototype which would help analyse these issues.

This chapter outlines problems in the current definition of MVC. It is not directly related to software usability, but acts as an input to the MMVC framework (next chapter), which does relate to usability. The discussion of MVC here is based around a timezone application — "MARCO" — which lets users display the time in a current location, and choose and edit locations (Figure 9.1). The system contains several domain models and has been developed using the basic conventions of MVC. Though not industry-scale, it does contain enough classes to examine the key issues in building multiple-model MVC applications; including abstract classes, there are 6 models, 21 views, and 8 controllers.

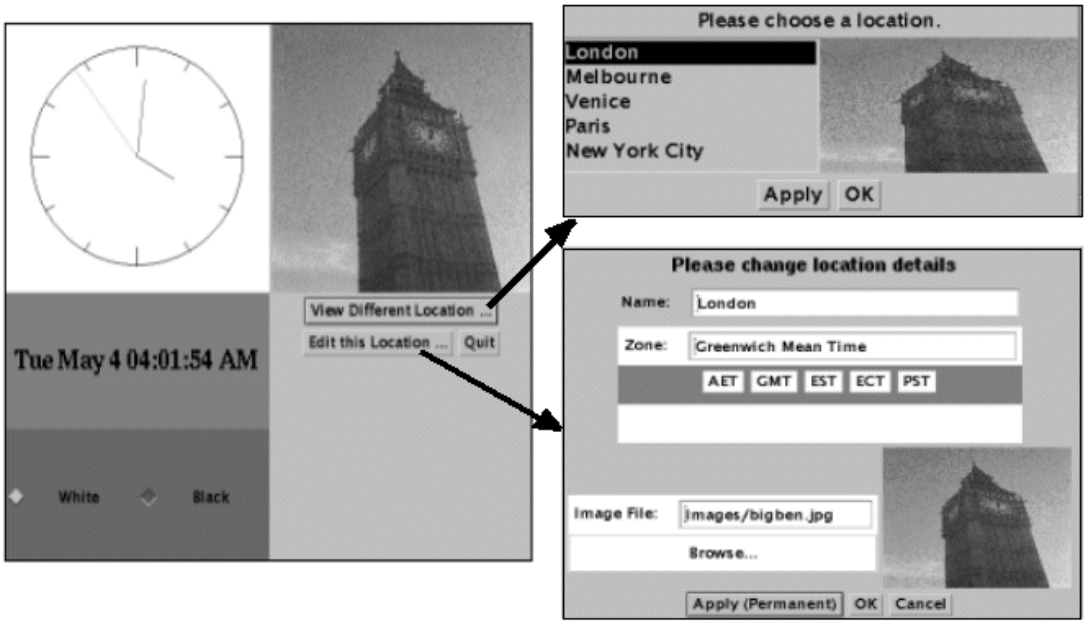


Figure 9.1. MARCO screenshots: "Focus View" of current location and time, Location Browser, and Location Change form (see Section 9.4.

MVC itself is overviewed in Section 9.2 and aspects of MVC in need of clarification in Section 9.3. The MARCO application is illustrated in Section 9.4 and its design shown in Section 9.5. Key lessons from MARCO's development are summarised in Section 9.6. Section 9.7 discusses MARCO and the implications for development of MVC applications.

9.2 MVC overview

MVC is a reference architecture, i.e. it may be viewed as a high-level architectural pattern. It follows a common theme among user-interface reference architectures: the separation of functionality from presentation. User-interface classes typically present domain classes to the user, enabling the user to view and control the domain. One benefit is the possibility of representing the same domain information in different ways. Designers can therefore tailor the user-interface to fit certain tasks and user characteristics.

The Model-View-Controller paradigm suggests three class categories ("stereotypes" in UML terminology) [130]:

Models provide the core functionality of the system and relate to the underlying domain (e.g. a car object).

Views present models to the user (e.g. a plan view of the car). There can be more than one view of the same model.

Controllers control how the user interacts with the view objects and manipulate models or views (e.g. an object enabling the user to change a car model using its plan view). There is usually a one-to-one relationship between views and controllers.

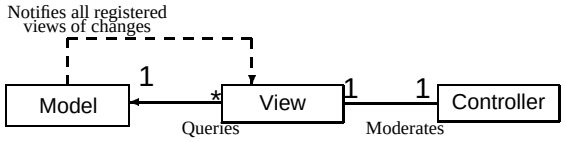


Figure 9.2. Static class diagram showing typical single-model MVC configuration. The view contains a direct reference to the model and the view and controller keep references to each other (the controller can obtain a reference to the model via the view). The model does not directly reference the particular views; instead, it sends change notifications to all views which have registered themselves with it. (the solid arrow indicates uni-directional association).

Figure 9.2 shows the relationship. Models do not directly call views. Instead, each view registers itself with its model, and the model notifies all registered objects whenever it is updated. This lets a developer add or change views without altering the model. It also ensures that views are synchronous because each view reflects the same model state.

This section focuses on one particular model in the timezone application: the clock model. This model runs in a separate thread, continually updating itself. Whenever it updates, the digital, analogue, and status views are notified, and they reflect the change by moving the hands or changing the displayed text within their panels (Figure 9.1 shows digital and analogue clock views).

The digital view has a panel which is created and monitored by the digital view's controller. When the controller detects the user pushing a timezone button, it sends a message to the clock model to set itself to the new timezone. Again, the model will notify its views that it has changed. For illustration, the digital

controller also allows the user to alter the colour of the view. This is a different kind of event because it will change only the digital view; the model will not be notified and the analogue view will therefore be unaffected.

The static class diagram is shown in Figure 9.3(a). View and ClockView encapsulate generic viewing capabilities. They ensure that each concrete view holds a reference to the clock model. They also define several methods which let concrete views initialise themselves, register with the model, create a controller if necessary, and redraw themselves whenever the model updates.

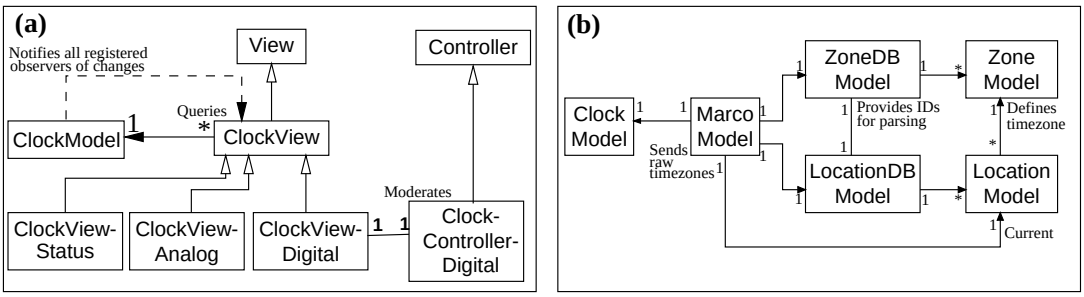


Figure 9.3. MARCO design: (a) Static class diagram of ClockModel and its. views and controller. (b) Static class diagram of all domain models. Each of these models has its own views and. corresponding controllers.

MVC's efficacy is demonstrated by its widespread influence in the design patterns arena. Gamma et al. [88] explain how `Observer`, `Composite`, and `Strategy` relate to Smalltalk MVC. Views are `Observer`s of models. Views are `Composite`s containing their own information as well as sub-views. Controllers define a `Strategy` which moderates the behaviour of Views. A pattern specifically about MVC has also been documented[34], but it mainly covers the relationships between model, view, and controller, and only describes a single-model example. The next section raises questions about MVC in the context of larger or more complex software.

9.3 Aspects of MVC in need of clarification

9.3.1 Role of controllers

The view and controller are supposed to create and maintain the user interface, track user events, and update the model when appropriate. If the designer understands the model, and knows how the user interface should behave, it is quite straightforward to construct a hybrid view-controller class. Indeed, the Document-View architectural pattern — exemplified by the Microsoft Foundation Classes — does exactly this, collapsing MVC's view and controller into a single class. PAC's presentation class does likewise [61].

However, MVC calls for separate view and controller classes in order to provide more flexibility to the user interaction. With a modern user interface toolkit, the distinction loses some of its relevance [44] and

a common question about MVC is whether the distinction is really necessary [193]. In particular, there is usually no need for the programmer to send events to the appropriate widget; the external environment ensures that the appropriate widget will receive events. Separating visual properties from event-handling capabilities can also be difficult, because modern widgets encapsulate appearance and behaviour. Nevertheless, there are still some good reasons to separate view and controller. It can be helpful to add new buttons, keyboard shortcuts, and so on, without directly changing the view. The key issue is creating a workable policy which determines how the class types are separated.

9.3.2 Updating models

Consider an application where the user can select several file icons at once. In this case, each icon is a view of type "Icon" on a File model. Another model, perhaps one representing a new directory, will store the list of File models. In MVC, this could be difficult to achieve. The icons present views of File models, but the controller of these views enables the user to change Directory models, not File models. This deviates from a basic rule of MVC, that a view-controller pair is associated with one model. It is not hard to see why this rule is in place. Allowing every controller to interact with any model in the application (or any controller) would create excessive coupling, since there can be many different controllers for each model. The coupling would lead to unmaintainable code, and make it impossible to reuse the components.

MVC's tendency to split control across all three component types has been noted previously [61, 203]. Indeed, a variation called MVC++ has been devised, in which controller classes mediate between views and models, and only controllers can interact with other M-V-C groups [117]; PAC takes a similar approach. The Visualworks Smalltalk framework provides a broadcasting protocol enabling messages to be sent down the visual component structure, which can help reduce direct coupling of views. This framework also provides specialised model classes (e.g. AspectAdaptors) which can help views to access models contained within other models [109].

The present work aimed to investigate the applicability of MVC itself to systems with more than one model. For a practical system, it would usually be most productive to rely on a language-dependent framework. However, doing so here would restrict options and make lessons less generalisable. Thus, MARCO uses a relatively portable event-generation mechanism can be used to overcome the scaleability problem.

9.3.3 Reusability of components

Reusable component technology has become a widespread issue in software engineering in recent years [129]. Developers are hungry for autonomous, cohesive, dynamic packages which can be incorporated into target systems without having to devote large amounts of time re-coding or re-configuring. Component reuse was one of the benefits touted for MVC a few years ago, but this referred to buttons, popup-menus, and so on. These are widgets provided by most modern toolkits. Whether or not the MVC framework

provides a better way to use these widgets may still be the subject of debate, but clearly modern components should do more than just act as widgets.

To facilitate greater reuse in MVC, packages can be created, each consisting of one model class and their associated view and controller classes. The problems with scaleability and control discussed above suggest that classes may not be independent enough to achieve this goal. Models interact with other models. Views add and remove subviews — which are often views of different models. Controllers receive input through one view, and it may often be tempting to permit them to alter other models. Thus, the challenge for MVC component developers is to create autonomous Model-View-Controller packages. MARCO uses domain-specific Model-View-Controller packages, and it is also possible to create domain-independent packages of this nature, as discussed in the next chapter.

Java's Swing library is a step in this direction, with a few semantic models, such as Tree, Table, and List models with corresponding views and controllers. MARCO does likewise for domain-specific components, such as Clocks and Locations. The program provides an insight into the construction of components in a language-independent manner, by: (a) being based on the more primitive AWT library, rather than Swing, and (b) avoiding the use of Javabeans, which facilitates component reuse. Although these are useful tools, their absence improved the proof-of-concept argument. Furthermore, components such as Tree work well in isolation, but they still leave open the question of combining components in a reliable and flexible manner. The next section considers how higher-level components — such as databases of models — can build lower-level components.

9.4 Multiple-model MVC application in action

The sample program is called MARCO (Multiple-Agent Real-time Clock Objects [139]) and was developed to clarify the aspects of MVC discussed above. This enabled MMVC to be developed with a solid understanding of the architectural pattern being used. The program is implemented in Java 1.1.7.

The application is a clock which can show the time at various locations in the world. Whereas the single-model application discussed above "hard-coded" the locations, MARCO reads initial locations from a file and the user can change the location database through the interface provided. A fixed database stores the domain of timezones for the locations along with common names (e.g. "Greenwich Mean Time").

There is a model called MarcoModel which represents the overall system state, similar to the ApplicationModel concept in VisualWorks [109]. From this, there are three top-level "MarcoViews", i.e. views of MarcoModel. Each can be run as a separate program. Each MarcoView is composed of views of lower-level models. In fact, it is also possible to run all three overall views at the same time, and the sub-views of each will stay synchronised with each other.

1. MarcoViewFocus This view (Figure 9.1) contains two clock views, an image of the location, and buttons which let the user change the location details or switch to another location.

When the user clicks on `View Different Location`, a Location Browser is loaded. The user can browse through locations, see their images, and choose a new location. Whenever a new location is chosen, the main view updates its clocks and image. There is no limit on the number of location browsers which can be open at once.

By clicking on `Edit this Location`, a Location Change Form is produced. This lets the user modify the location's name, timezone, and image. When a user changes a location's details, the modifications propagate to all other views. When an image is updated on the change form, for example, the image will also change in the main view and all location browsers in which that location is selected. When a timezone is updated, the two clocks update. If the user chose to edit the current location, then selected a new location with the browser, the change form would still relate to the initial location. Thus, changing details on the form would have no effect on the main view. The user can have any number of browsers and change forms open at the same time. All are kept synchronous.

2. MarcoViewBackground This view (Figure 9.4) shows a digital clock and lets the user change timezone (instead of location, to demonstrate flexibility). It is a toplevel window with just the digital clock view and the zone-choosing component which is contained in the Location Change Form.

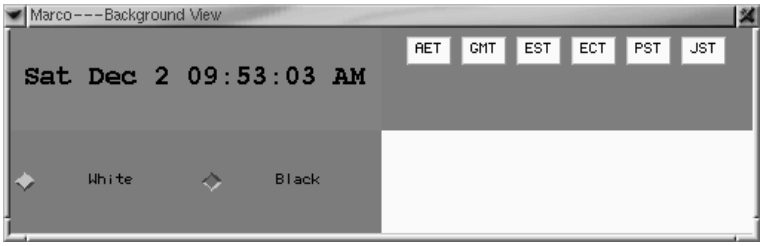


Figure 9.4. MARCO screenshot: Background view, enabling user to select Zones rather than Locations.

3. MarcoViewStatus This is a more technical view, showing internal status strings for each object and could be used for development purposes (Figure 9.5).

9.5 Design overview

9.5.1 Models

The MARCO system consists of six domain model classes, whose relationships are shown in Figure 9.3(b). The models are:

MarcoModel An overall application class which exemplifies the `Mediator` pattern [88]. It stores a ClockModel, a ZoneDBModel, and a LocationDBModel and also a reference to the current LocationModel. It fulfills the vision of component reusability by demonstrating how the other components can be brought

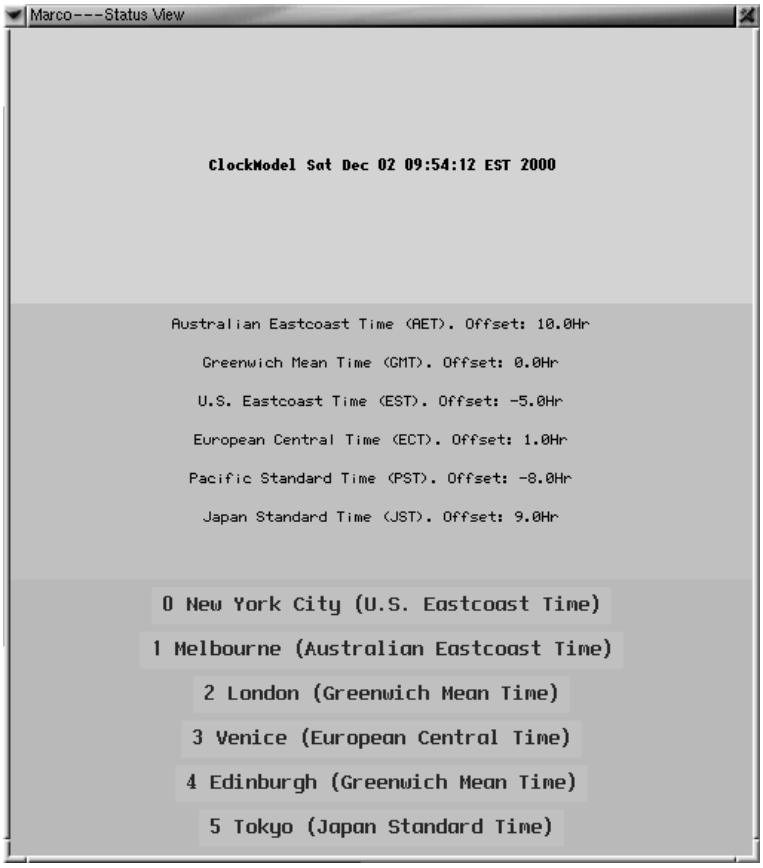


Figure 9.5. MARCO Screenshot: Status view, enabling a programmer to check internal state of Models (by showing output of Model's toString() method). The view can be used to watch internal states change by creating it in parallel with other views, e.g. the main view MarcoViewFocus.

together to interact with each other. The ClockModel does not know about the other classes; nor do they know about the ClockModel.

ClockModel A clock with an associated timezone which continually updates itself. It uses the TimeZone library class to calculate appropriate time.

ZoneModel A timezone, common name and Java TimeZone library ID.

LocationModel A location with a name, image, and associated ZoneModel.

ZoneDBModel and LocationDBModel Classes capable of storing and manipulating groups of ZoneModels and LocationModels (the list is stored as a Vector class).

Component autonomy is ensured by state information being stored only in MarcoModel; the databases, for instance, do not store the current location or zone. All models extend Java's Observable class, which provides the behaviour required to maintain a list of observers and notify them upon changes to the model (the class could easily be created for another programming language).

The initialisation of models is shown in Figure 9.6. MarcoModel requests both databases to parse the files containing zone and location information. It becomes an observer of these databases, explained in Section 9.6 (Problem 5). It then sets the current location to the first item in the location database, and starts the clock, with this location's timezone. The clock maintains independence from other packages by accepting only standard Java TimeZone objects, rather than ZoneModels.

9.5.2 Views

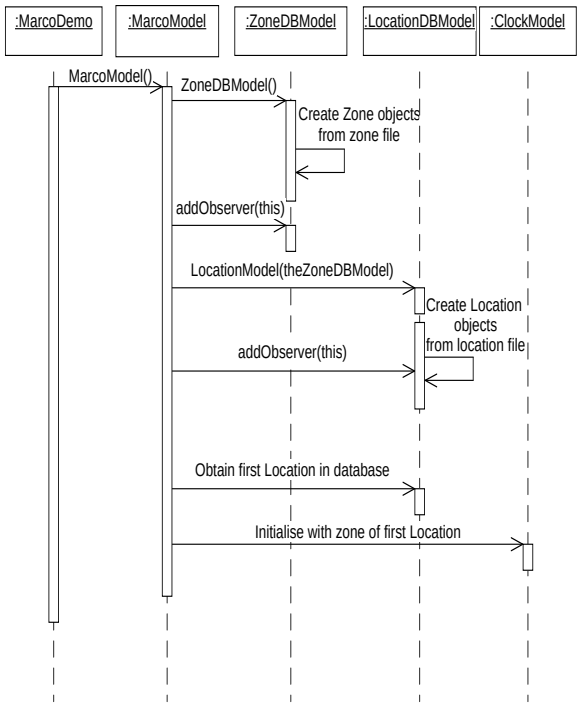


Figure 9.6. MARCO design: Sequence model showing initialisation of major models.

There is an abstract View class which does the following: registers itself with the model and stores a reference to the model; defines an update method which (a) checks that the correct object has notified it and (b) calls the (abstract) draw method, for objects to inspect the model and redraw themselves; provides an empty method to create the controller, which subclasses can override if they require a controller; contains

convenience methods setVerbosity() and getString() which enable subclasses to easily obtain strings which portray the model's states at varying levels of detail.

The structure of Views follows the Composite pattern [88]. When a View is created, it constructs its own widgets and also adds sub-views. The sub-views look after themselves. Thus, the clock views inside the main Focus View both observe the Clock model and update themselves accordingly. The Focus View itself does not have to observe the Clock.

View does not contain a reference to the model, because the model type is different for each subclass of View. Instead, there is an abstract view class for each model, e.g. ClockView. This class stores a reference to the model, provides a constructor which sets the model, and can define the convenience string-handling methods. Model-specific views can also perform miscellaneous functions specific to the model type. ClockView, for instance, overrides the update method to impose a minimum number of seconds between each redraw.

The concrete views have one or more widgets which can be obtained via public methods. For instance, the timezone selection view provides one panel showing buttons for each zone, and another status bar label which shows the zone's common name whenever the mouse hovers above a button. A developer using this class has the freedom to show only the buttons, or the buttons with the status bar, and the two widgets can be placed in separate screen locations.

9.5.3 Controllers

It was decided to explicitly distinguish between controllers which control through user interaction with specific views and those which are generic and control a model directly. The one class in the second category is the Location Change Form. It is initialised with the model as a parameter, and disposes itself when the user hits the "OK" or "Cancel" buttons. Its independence makes it easy to add to existing views. The controller for the location status view enables the user to produce the form by double-clicking on the status text. This will generate the location form, and when the user changes location details, the model will be altered, and the original status view will update itself.

In this implementation, controllers which relate to views have several responsibilities. First, they implement Java's event-handler interfaces to listen for appropriate events, e.g. the zone selection controller listens for occasions when the user rolls the mouse over a zone button and tells the view's status bar to update. Second, views delegate construction and maintenance of button panels to controllers. Third, controllers can broadcast semantic events (e.g. zone changed), to objects who have informed the controller that they are interested in these events.

Following the initialisation of models (Figure 9.6), MarcoViewFocus is constructed with its model. It stores the model and begins to observe it. This is standard procedure for all views, embodied in the View superclass constructor. Another procedure inherited from View, `initialise`, begins by calling

`makeController`. In the case of MarcoViewFocus, this is left blank because no constructor is required. `initialise` then calls `constructInitialView` to create a digital clock view, an analog clock view, and a location image. The digital clock view creates its controller. It then constructs its own initial view, consisting of a label to store the time on as well as the control panel obtained from its controller. Finally, it calls `draw`, which shows the current time. `MarcoViewFocus.constructInitialView` creates the analog view and the location image in this same way. It then updates itself by calling its own `draw` method. Since the subviews update themselves, this method is very small; all it does is update the window title.

Incidentally, it might seem odd to have an `initialise` method — why not use the constructor? The answer is that `initialise` creates a controller. Since the controller will immediately store a reference to the view, it cannot be called from the view's constructor because the view reference will not yet be valid. Buschmann et al. [34] also use an initialise method for this reason.

9.6 Lessons learned

The purpose of MARCO was to clarify MVC before developing the MMVC framework. Therefore, it was important to clearly capture the lessons which were learned from the development process, in terms relevant to future MVC applications. Therefore, the key design decisions are documented in this section. With only one application to draw on, it is inappropriate to represent the findings as patterns. They are instead, an analysis of encountered problems and solutions which is intended to highlight the overall implications of using MVC.

• **Problem 1:** The view and controller can be separated in different ways — see Section 9.3.1. Which is appropriate here?

Solution: Controllers implement the callback methods which handle events on the view's widgets. Some coupling between view and controller is still necessary, because the views must configure the widgets with the appropriate listener. For instance, the Location Status View must inform its display panel that the location status controller will listen to button-clicking events. However, the level of coupling is non-critical, since this is the only situation in which the controller is referred to. Furthermore, most event-listening occurs on button-panels, and these are created by the controller.

This approach splits presentation across two classes, which could lead to confusion. To minimise complexity, views provide some methods for the controller to perform output, e.g. the zone database view provides methods to set the status bar according to a string or a ZoneModel parameter.

Java's event-handling mechanism provided a convenient way to use controllers — to ensure that controllers implement the appropriate event-handlers. For efficiency purposes, the Java 1.1 toolkit requires specific event listeners, e.g. a mouse listener, a text component listener. This forces some coupling between view and controller, because it means the view must add the appropriate listener to each widget.

The Location Browser is a controller with no views, but could also have been implemented as a view

with text fields, connected to a controller which knows how to handle them. This would have been rather contrived, however, and led to a very tight coupling between view and control. Clearly, the form's purpose is to control the model. Therefore it was reasonable to abandon the typical MVC pattern of one-to-one view-controller relationships.

Although this approach reduces coupling between the components, it is still important to note that the roles do overlap. The benefits of separating view and controller are questionable, given that both components are coupled to the model (the view by reading, the controller by changing it). The example of Location Browser also suggests that the distinction can be rather arbitrary.

• **Problem 2:** Each model-view-controller package should rely on as few other packages as possible.

Solution: The relationship among views and controllers is aligned with their models. That is, if a model M_1 depends uni-directionally on M_2, then views and controllers of M_1 can use the M_2 package, but not vice-versa. This means that a package of a model and its views and controllers has its degree of independence determined by the independence of the model alone.

To promote independence of models, class associations should be uni-directional where possible. In Figure 9.3(b), ZoneModel and its associated views and controllers ("the Zone package") is completely independent. The database of Zones, ZoneDBModel, depends only on ZoneModel, so the pair of classes could be reused, for example, in travel-related software. In fact, there are several combinations of packages which could easily be combined and reused: {Zone}, {ZoneDB, Zone}, {LocationDB, Location, ZoneDB, Zone}, {Clock}.

Alternative Solutions: One alternative design would eliminate the database classes and use MarcoModel to directly store lists of LocationModels and ZoneModel. This would force MarcoView subclasses to allow users to manipulate the lists. However, this is functionality which would be desirable in any application using Location and Zone, and should consequently be separated from the application, hence the need for database classes.

• **Problem 3:** How can views and controllers present information relating to one model and simultaneously enable a user to change another model? For example, the Location Change Form controls a LocationModel, and the user selects the zone from an inset Zone Database View. The dilemma is that the database view presents one model, but user events on the database view affect another model. If the zone database controller directly changes the location model, or instructs a location controller to do so, a bi-directional dependency is introduced.

Solution: Objects can generate semantic events to signal important changes, and this avoids them having to notify specific classes.

The handling of the above example is shown in Figure 9.7. When it was initialised, the Location Change Form registered itself to receive semantic events from the inset zone database view. When the user clicks on a button on the zone database, the zone database controller catches the event. It generates

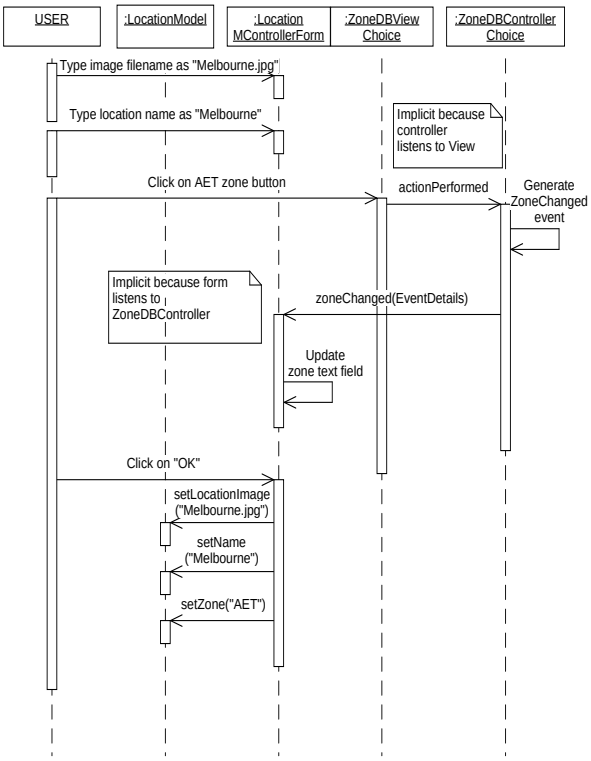


Figure 9.7. MARCO design: Sequence diagram showing user editing Location (see Problem 3).

a `ZoneDBEvent`, a newly-defined event type which stores the view that was chosen. A callback method in the location form receives this event, and updates the zone text field beside the database view (if desired, the form could immediately change the underlying model).

Controller classes maintain a list of classes which have registered for the event, and when the user implicitly requests the event via the user-interface (e.g. clicking on a zone button), a method is called to loop through each registered class and then call its callback method (e.g. `zoneChanged(ZoneDBEvent theZoneDBEvent)`).

Semantic event generation is one important way to scale up MVC applications. Events can be refined in more abstract terms as they move upwards towards higher-level classes. This helps to reduce complexity. Reusability is also facilitated; in this example, the zone database view does need to know the specific classes which use it.

Semantic events are used in various frameworks, such as ActiveX, Javabeans, and the Business Object

Component Architecture (BOCA) [70]. Often, these components effectively represent a view, a model, or both, and event types could be a user-interaction or an internal change. Also, widgets typically use this notion to some extent; as well as allowing callbacks for primitive user events such as button-clicks, many widgets can notify the program of such higher-level events as an item being added to a list. In the context of multiple-model MVC, semantic events are beneficial because they reduce direct references between objects.

Alternative Solutions: It is not feasible to use the existing Java event and listener classes, because the events do not carry the semantic information required, e.g. the ZoneModel which the user requested. This is why new classes were required, as well as controllers to keep lists of listeners. However, the code to do this is minimal.

It would also be possible to make controllers Observable, and let interested parties implement Observer to watch for changes. This is really another version of registering for events, but would be an abuse of the Observable class, since user events do not really constitute changes to the controller. Semantic events are more flexible, as there can be more than one type of event, and they are conceptually a more logical way to handle semantic-level user requests.

• **Problem 4:** Some models and views have their state defined by one or more models. In the case of the location browser view which enables users to switch between LocationModels, the present state can be defined by the LocationModel which the user has most recently selected from the list (a property of the browser view). As the user clicks on list items, the corresponding image view must change to reflect the current model. A similar situation arises for MarcoViewFocus, whose image view must change when MarcoModel's current LocationModel changes.

Solution: When a container view includes a view of a model class, and the particular model object might change, a view of each possible model object is created upon initialisation and whenever the possible set of models changes. Whenever the model object changes, the view is cached in memory and simply needs to be displayed.

For example, when the LocationModel database changes, a list of views is created, one for each model in the database. Whenever the user switches models, the pre-existing view is replaced by the view of the selected LocationModel.

This approach has the disadvantage that view objects are updated whenever their models change, even if they are not visible. A more sophisticated technique — perhaps based on Gamma et al.'s Proxy pattern [88] — could prevent this by ensuring views only update themselves when they are visible.

Alternative Solutions: Perhaps a more obvious solution is a "lazy" approach: simply create a view of each model whenever it is required. This would certainly be appropriate if the user rarely changed the current model, or there were many models. This approach uses less memory than the previous method, but will be slower. To improve speed, a set of created views could be stored. However, this would be similar to the solution described above.

Alternatively, it would be possible to create a new state model on initialisation, and create a single view, because it would update as the model changes. However, the model would be a copy of the real model in the database, rather than a reference, and this means that when a user tried to change the current model, they would be changing a copy of it; the real model in the database would not be updated. This complication could probably be worked around, but would pose a large risk of synchronisation errors.

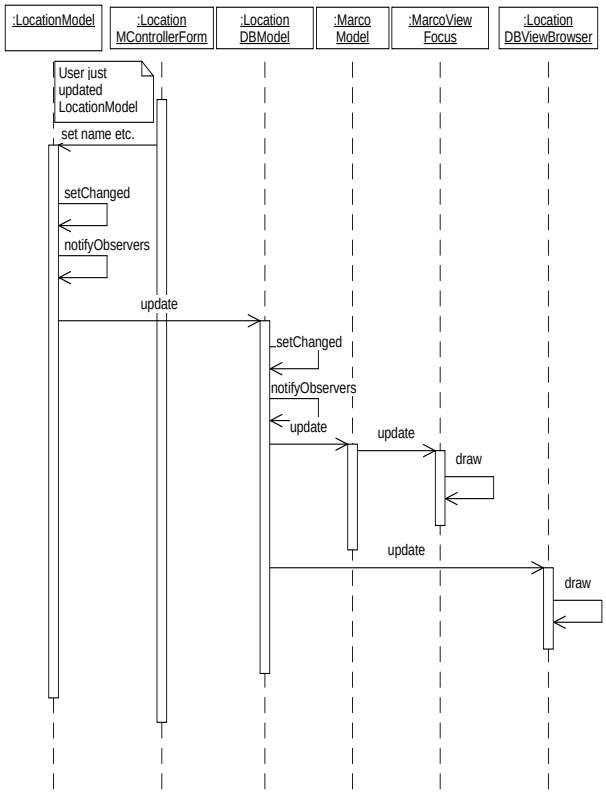


Figure 9.8. MARCO design: Sequence diagram showing updating of views to reflect Location details change (follows Figure 9.7).

• **Problem 5:** Some models reference other models, and need to know when they have changed. For example, the overall application model, MarcoModel, needs to know when the current Location changes zone, so it can inform the clock. However, to avoid a cyclic dependency, the changed models should not directly communicate with the model which references them.

Solution: When a model stores a reference to another model, it registers itself as an observer of that model.

Figure 9.8 shows what happens when the user changes details on a Location Change Form. It follows immediately from Figure 9.7. The Location database has registered as an observer of each of its Locations, and therefore knows that the LocationModel in question has been updated. MarcoModel, in turn, is observing the LocationModel, and also updates. As with the standard MVC definition, MarcoViewFocus notices the change to its model and also updates (it redraws the window title). All other views of the location database will also update, such as any Location Browsers which are open.

This approach is helpful to developers of large systems where high-level models depend on lower-level models, and the changes must be propagated throughout. It creates some additional overhead because higher-level views need to update, but it would be possible to create views which only update if the features they are displaying have changed.

• **Problem 6:** Each location has an associated image. Where should the image be stored?

Solution: The image is stored as an attribute of LocationModel. This may be surprising because one might think that views should store images. However, a view only knows how to render information stored in its model. Instances of LocationViewImage differ only in their model (and possibly some detailed aspects like scale). In contrast, LocationModels vary across all attributes — name, image, ZoneModel.

Alternative Solutions: If there was a large amount of visual information stored in the LocationModel, it might have been helpful to separate it from the semantic details by storing it in a separate LocationModelVisual class, as an attribute of LocationModel.

9.7 Discussion

The multiple-model MVC program was developed quite smoothly using some of the concepts described in this chapter. Initially, it was anticipated the program would act as an illustration of the overly-complex nature of MVC, and that another architectural pattern may have to be adopted for the consequential detailed design framework. However, the result was quite reasonable, with a good degree of independence among packages and a manageable policy of combining packages together. The design's generality is evidenced by the fact that it has been implemented in a fairly standard language (i.e. Java), and relied on the kinds of libraries which are available for most modern languages.

Some inherent weaknesses with MVC do remain. It is difficult to separate views and controllers in a totally clean manner, because modern toolkits usually mix output and input. Even though there will always be some coupling, the separation is likely to be worthwhile in many cases because there are always going to be different ways of manipulating the same view.

Another weakness is performance. The registration mechanism has been extended here to allow models to observe other models. Furthermore, the semantic event-generation concept has also been introduced to cope with multiple packages. Both of these patterns will cause some performance degradation. The maintenance of view lists does likewise, although optimisations — like an image view cache — are possible.

The program is larger than many previously-documented MVC applications, and therefore provides some insight about how to integrate different model-view-controller packages and design for reuse. However, with five domain classes, it is hardly an industry-sized system. This leads to the question about whether the concepts introduced would scale further. As long as the software could be broken into cohesive subsystems, it seems feasible to suggest that large-scale systems are able to be developed with the ideas described here. In particular, this would be possible by defining clear policies for splitting view and controller, applying semantic events, ensuring that models store other models as references rather than as copies, and having models register to observe changes on other models.

Future work should aim at establishing the validity of these ideas by applying them to new applications. It would also be valuable to enhance the framework by applying more flexibility. For instance, a subview might adapt to the colour and layout scheme of the view it is inserted into. The work also suggests the need for a code-generation tool, since a large amount of code in the program is of the trivial-but-necessary variety, and could easily be automated. Such a tool is described in Chapter 10. Finally, generalisability was improved by avoiding reliance on any particular language or framework. However, it would still be useful to investigate whether the findings here could assist with development under an existing framework such as Java's Swing.

The shortcomings of MVC have been studied and suitable modifications have been developed to handle MVC in a more scaleable manner. These lessons are applied in the next chapter, in which several generic tasks are implemented under MVC.

Chapter 10

MMVC: A Framework arising from the Implementation of Generic Tasks in MVC

10.1 Introduction

A list of generic tasks was generated in Chapter 8. The list of generic tasks, and the brainstorming study which followed, provide some evidence that the generic tasks are useful as a high-level design tool. As Figure 8.2 showed, high-level design is not the only development activity generic tasks entail. The other activity is detailed design, where generic tasks may be used to provide supporting tools.

The clarification of MVC in Chapter 9 was also necessary to provide detailed design support. In this chapter, the generic tasks concept of Chapter 8 is combined with the MVC results of Chapter 9 to illustrate how generic tasks may be implemented under MVC. In doing so, the aim is to construct a reusable framework which reconciles usability with other software attributes. This fits with the second aim of this thesis, namely reusing detailed design features that have contributed to usability. To the extent that generic tasks are a design feature, the work here also touches on the third thesis aim, to combine high-level design reuse with detailed-design reuse.

The commentary on applying MVC should in itself be interesting for software developers, but the focus here is on the way in which a detailed design framework can support usability. The framework addresses the needs of developers by showing them how to support generic tasks in a usable manner.

The framework in this chapter is called “Multiple Model-View-Controller” (MMVC). It is called a “framework” because it goes beyond patterns to include reusable components and a basic source-code generation tool. Experience with MARCO (Chapter 9) indicated that reusable components would be valu-

able, hence the opportunity has been taken to explore how they can further facilitate reuse of user-centered design. A source-code generation tool also helps by producing base code for concrete Models and Views.

A key inspiration for this work is Ralph Johnston's discussion of patterns and frameworks. This view can be summed up with the title of his 1994 paper, "Frameworks = (Components + Patterns)" [123]. As discussed in Section 3.2.2, Johnston argues that patterns can help to document reusable software, and that reusable software is actually written with usage patterns in mind. Thus, MMVC is used to explore how the synergistic interaction of patterns and reusable components can be harnessed to improve usability.

Initial ideas for the framework came from MARCO, and it was then developed alongside an example software product, Prescribe [142]. Prescribe is implemented using several of the lessons from MARCO, as well as the reusable components which also relate to lessons from MARCO. It implements several generic tasks, and therefore demonstrates how a software framework can assist detailed design of applications which support a particular task set. This would enable usability principles to be encapsulated in tools used by programmers, i.e. reusable components, source-code generation utilities, and detailed design patterns. The approach is obviously no substitute for usability expertise, but certainly contributes to greater emphasis on usability by those with a large influence on the final system. Consistency is also a great benefit, since generic tasks which are supported by the framework will be executed in similar ways and using a similar user-interface.

The Prescribe application is described in Section 10.2, which contains a walkthrough of the application, a list of the generic tasks which are supported, and a description of the software design. The framework is covered in Section 10.3, detailing the reusable components, the source-code generation program, and the pattern language. Section 10.4 is a discussion of the approach.

10.2 Medical Prescription Application

This section outlines Prescribe, the application developed alongside the MMVC framework. Prescribe is implemented in Java 1.2.2. It supports a several generic tasks, listed in Section 10.2.2 and makes use of the MVC lessons of the past chapter, as shown by its design description in Section 10.2.4. Since the MMVC framework concerns reuse of knowledge about generic tasks and MVC, Prescribe was used in the development of MMVC.

10.2.1 Application Walkthrough

Prescribe is an application intended to be used by medical staff within a hospital. Functionality was conceived by myself, based on my work experiences in performing observations of hospital-based medical staff and designing medical software. The application here, it must be noted, is an example which happens to be based in the medical domain rather than a prototype for an eventual working system. To use it for a real-life system, it would require greater attention to security and medicolegal issues, as well as additional

functionality such as patient management.

In hospitals, medical staff maintain a list of "Orders" — requests for drugs to be administered to the patient. When the doctor creates an Order record, they indicate drug, daily frequency, dosage, start and stop times [1]. Nurses administer the drug, recording time of administration. Prescribe automates these processes for a single patient.

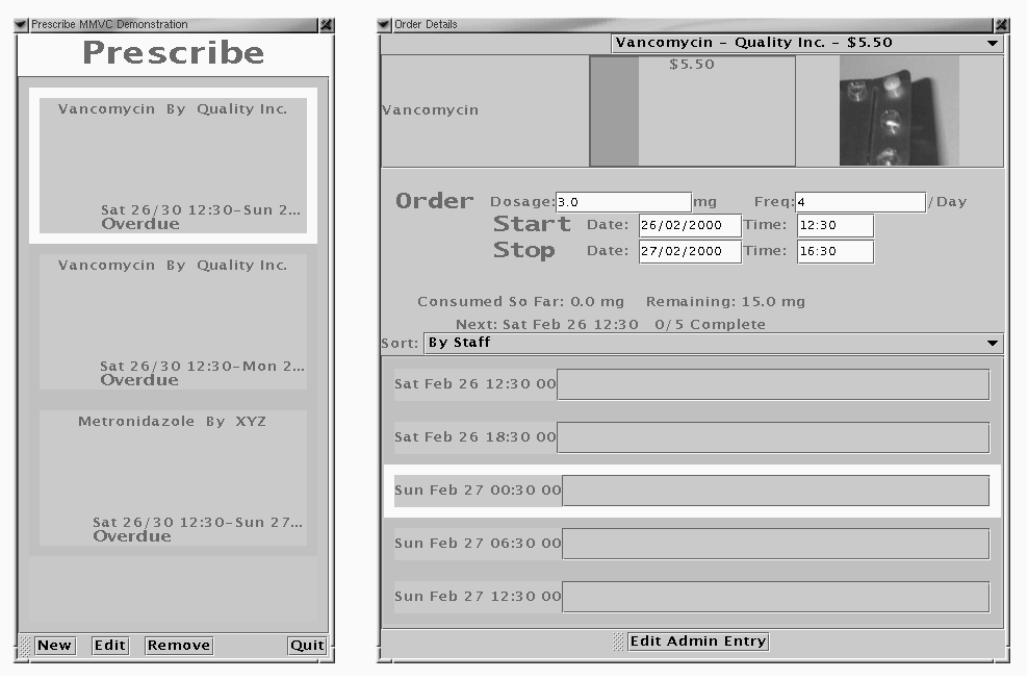


Figure 10.1. Prescribe screenshot: Main view of Prescribe (left) and View of Order (right), accessed via main view.

A screenshot is shown in Figure 10.1. The main Prescribe View shows three orders. The user has opened the third order, and can access the following Order information:

Drug The drug's name, image, and unit cost is shown. For demonstration purposes, the unit cost is shown in a bar indicating ratio to most expensive drug. The drug can be changed by selecting an alternative from the list control above the View of the currently-selected drug.

Order Parameters Dosage, frequency, start and stop dates are shown. The user can alter any of these values [2].

Dosage Statistics For this order, total dosage consumed and total dosage remaining.

[1] In reality, stop times are usually recorded at the time the order is discontinued.

[2] Dates and times should ideally be entered via a specialised date/time control. However, there appear to be no freely-available controls of this nature; it was decided that a straightforward text entry would suffice, provided that time and date fields were separated.

Administration Record The administration records are shown. These are automatically generated from Start, Stop, and Daily Frequency parameters. Each record shows expected time (according to Order parameters), actual time if it has already occurred, staff member (a proxy for authorising signature), and any comments about the administration procedure. Status information indicates next order, and number of complete orders. The user can sort this list by staff member name, by administration time, or by discrepancy between actual administration time and expected time.

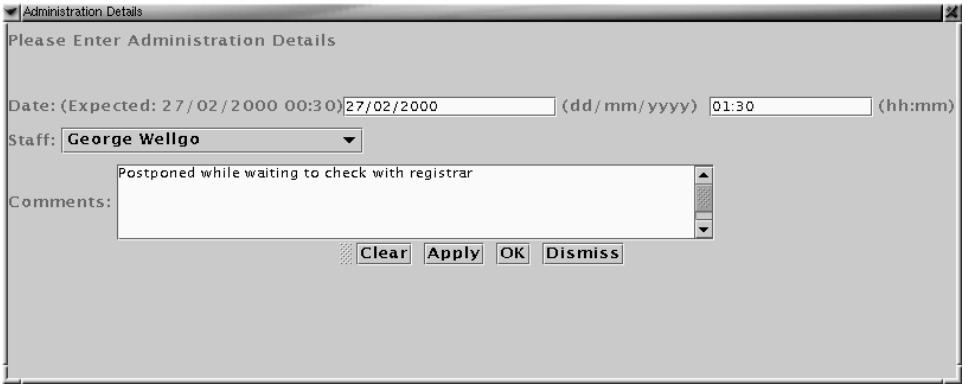


Figure 10.2. Prescribe screenshot: The Administration editor, allowing user to enter actual time of administration, staff member, and optional comments.

.

To set or update information concerning an administration, the user selects an administration record and activates the Edit button. This produces the dialog box shown in Figure 10.2. The main Prescribe View contains a list of Orders, each providing overview information. These Order Views indicate if an order is finished (i.e. all orders are complete) and also alert the user to orders with overdue administrations (i.e. an administration was expected before the current time and has not yet occurred). The main View allows the user to create, remove, or edit an existing order. As with the administration list, the user selects the record and then invokes a function by clicking on a button. The buttons in this case are “Add”, “Edit”, “Remove”. Note that this contrasts with the Administration list, which does not allow Add or Remove, and this is because the Administration items are automatically determined by the Order parameters; the user cannot alter the list.

The separation between Model and View enables any number of order Views to be open at once. Figure 10.3 shows the same order in two different windows. When the user alters details in one window, it will effect an update to the other window as well as an update to the Order View on the main Prescribe View.

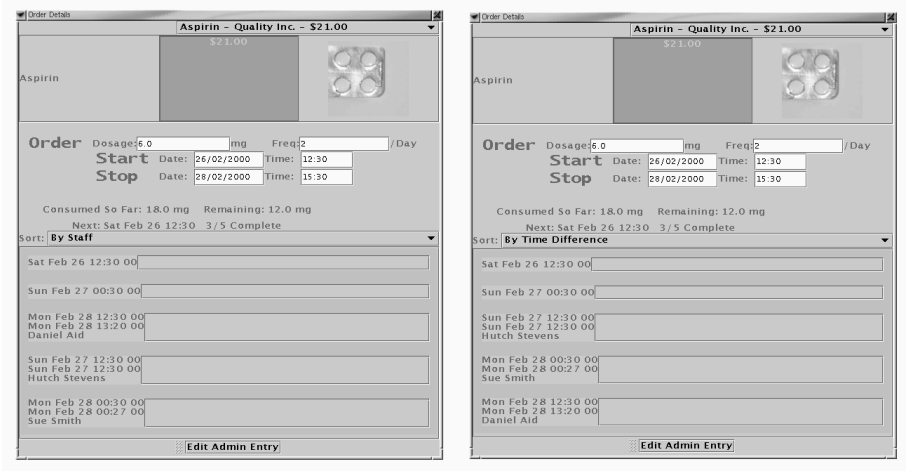


Figure 10.3. Prescribe screenshot: Single order rendered by two different detailed views. The view class is the same, but the sorting method is different for each view object.

10.2.2 Generic Tasks inside the Application

The application was conceived and designed with the intention of supporting several of those generic tasks captured in Chapter 8. They are:

Navigate The user can open an order dialog from the main Prescribe View, and an administration dialog from there.

Summarise The Order View and main Prescribe View both contain status information about the order and its administrations, e.g. total dosage consumed for this order.

Compare The user can compare two orders by opening both dialogs at the same time.

Enter Data Data about orders can be entered on the order dialog, and data about administrations can be entered on the administration dialog.

Select The user must select orders before editing or removing them, and administration records before editing them.

Add The user can add orders.

Remove The user can remove orders.

Replace The user can switch between different drugs within the order dialog.

An important theme is the manipulation of lists, suggested by Add, Remove, Select, and Enter Data.

10.2.3 Software Design Objectives and Relationship to MARCO

Prescribe builds on the concepts developed in Chapter 9. As will be discussed in Section 10.2.4, several design concepts have been reused in Prescribe. In addition, there were some modifications or enhancements which were suggested by MARCO:

Controller Incorporated into View The controller class was of questionable value in MARCO, with the distinction between View and Controller rather arbitrary. It has been discarded for Prescribe. This should not lead one to the conclusion that MVC has essentially been dropped. The View class in Prescribe is effectively a combined View-Controller, which renders and alters its Model class via a broadcast mechanism, just as with regular MVC. Trygve Reenskaug, the major figure behind the creation of MVC [222, 104], indicated, in the context of his recent OOram methodology, that he does not feel strongly about the View-Controller separation (p.253) [193] :

> *There are strong programming arguments for separating the model and the view. We find that many views are reusable by widely different models...The value of separating the view and the controller is not as evident. There are examples of views being associated with different controllers, but much of the same functionality could be achieved by suitable configuration facilities. (Italics added.)*

Reusable Domain-Specific and Domain-Independent Components As with MARCO, the design of domain-specific components provides a good degree of autonomy. For instance, a Drug Model and its associated Views could be reused in a pharmacology application. In Prescribe, the extent of reusability takes a step further with several domain-independent components. To facilitate the add/remove/edit tasks (suggested by generic tasks Add, Remove, Enter Data), there are reusable classes that enable lists to be maintained within a Model-View-Controller framework.

Tailored to Language/Libraries MARCO was deliberately developed without using Java's Swing components, which incorporates some concepts of Model-View-Controller, so as to study the language-independent issues of MVC. For Prescribe, the aim was to produce a coherent framework, as a proof-of-concept. Swing, which contains more powerful components than primitive AWT and better layout capabilities, was therefore adopted in Prescribe. It was found that Javabeans concepts were also helpful; thus, the Models in Swing are Javabeans, enabling external objects to receive notifications of their property changes (described in Section 10.2.4). This is a change from MARCO, which used the observer class.

10.2.4 Software Design

As the static class diagram (Figure 10.4, p. 203) makes apparent, the application consists mostly of Model and View classes. The Models follow the Javabean convention of "get" and "set" methods for each property.

For instance, OrderModel contains the following methods:

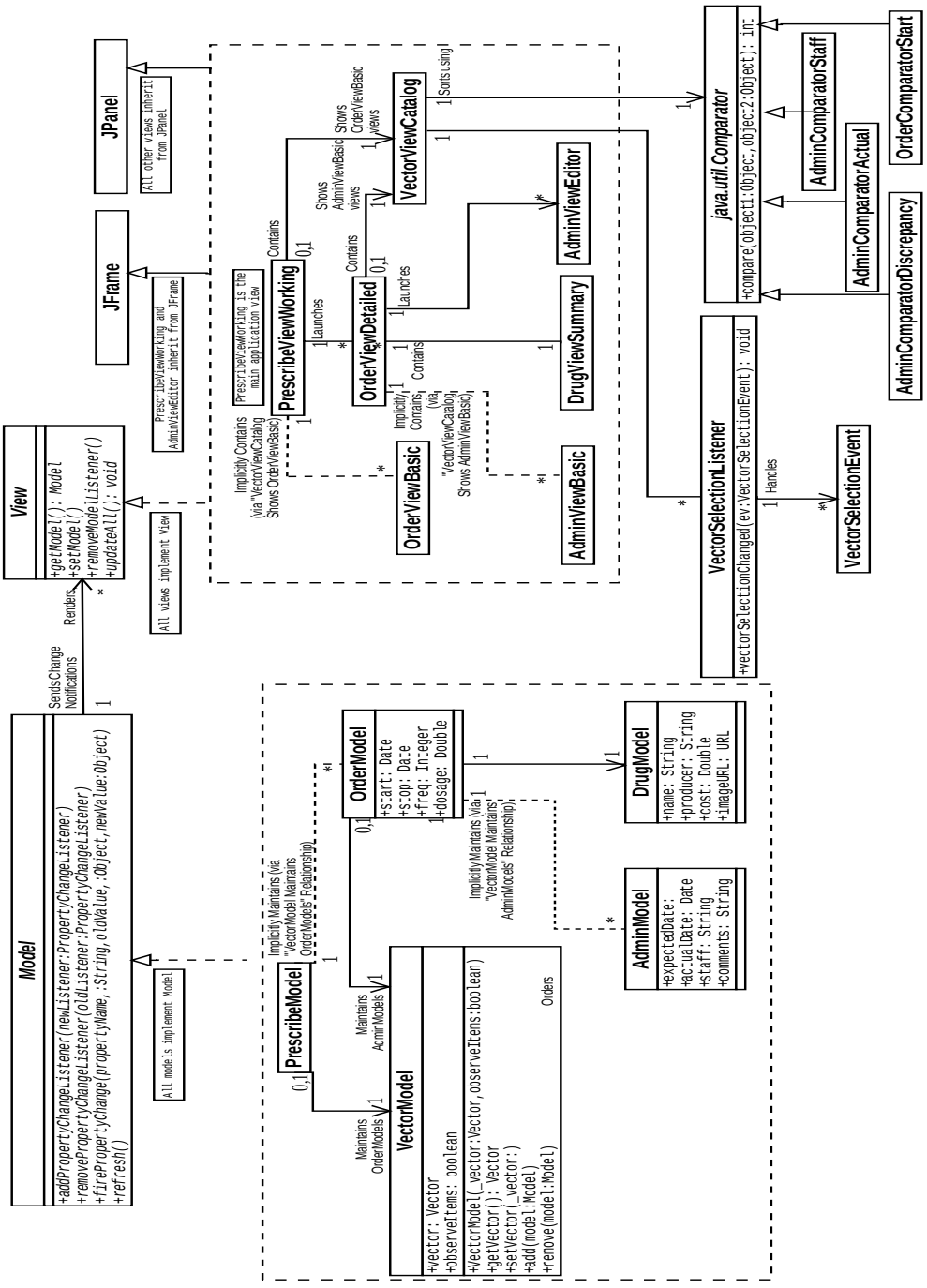


Figure 10.4. Prescribe design: Static class diagram

- `public void setDosage(_dosage);`
- `public Double getDosage() { return dosage; }`

This convention (which is not unlike the form of many implementations in non-Java OO languages) would enable a third-party tool to infer that the OrderModel has a ”dosage” property (which may have nothing to do with the internal representation of dosage).

The java.beans package supports the concept of properties, and is used by Prescribe Views. The Views have a setModel(Model) method which allows a caller to attach the Model. Not surprisingly, this method sets the View's Model attribute to the new Model. It also requests the Model to send notification whenever Javabean properties are updated. This is a more efficient way to deal with the broadcasting mechanism than was achieved with MARCO. The View knows which properties have been updated, thus it only updates itself rather than completely redraws whenever a property changes.

The sequence of events is demonstrated in Figure 10.5. In this case, the user is altering dosage on OrderViewDetailed, the editable Order View shown in Figure 10.1. The View's FocusListener notices the keyboard focus just left the dosage field. The FocusListener sets the Model's dosage (since it is inside the View, it has access to the View's Model). OrderModel.setDosage calls FirePropertyChange to indicate it is changing a Javabean property belonging to OrderModel. The object keeps track of listening objects via a propertyChangeSupport member, an instance of Java's java.bean.PropertyChangeSupport class. FirePropertyChange asks the propertyChangeSupport to alert listeners that the change has occurred and it specifies the affected property. This causes OrderViewDetailed and OrderViewBasic to receive change notifications, along with an indication that it is the “dosage” property which changed. Each View is aware of its respective components, and how they relate to dosage. They call updateDosage to respond to the changes.

All of the Models shown in Figure 10.4 interact with their corresponding Views in the manner just described. The VectorModel is particularly interesting, because it is a Model which stores a vector of other Models. PrescribeModel uses it to store the list of orders, i.e. the VectorModel contains a vector of OrderModels. VectorModel is complemented by one View class, VectorViewCatalog. VectorViewCatalog is capable of rendering a VectorModel. VectorViewCatalog's creator can inform the class how it would like the individual Models to be rendered, i.e. which View class to use. The PrescribeViewWorking View (Figure 10.1) contains a VectorViewCatalog showing the orders. When PrescribeViewWorking creates its VectorViewCatalog, it specifies that OrderViewBasic should be used to show the OrderModels.

A further feature of the vectors is the use of Java's Comparator class, which enables a flexible sorting mechanism. At run-time, the user can provide the VectorViewCatalog with a comparator, which VectorViewCatalog will use to sort the list. The View is immediately updated when the comparator changes. The comparator belongs to the View rather than the Model because it is related only to what the user sees. It also means a user can maintain two simultaneous Views of the same Model, with data sorted in different ways. Section 10.3.1 expands on the Vector classes.

Once the Vector concept is understood, the domain Models are quite straightforward. PrescribeModel tracks its OrderModels via a VectorModel. OrderModel contains a DrugModel and a VectorModel of AdminModels. Each Model has some Javabean properties. These can be regular values like OrderModel's

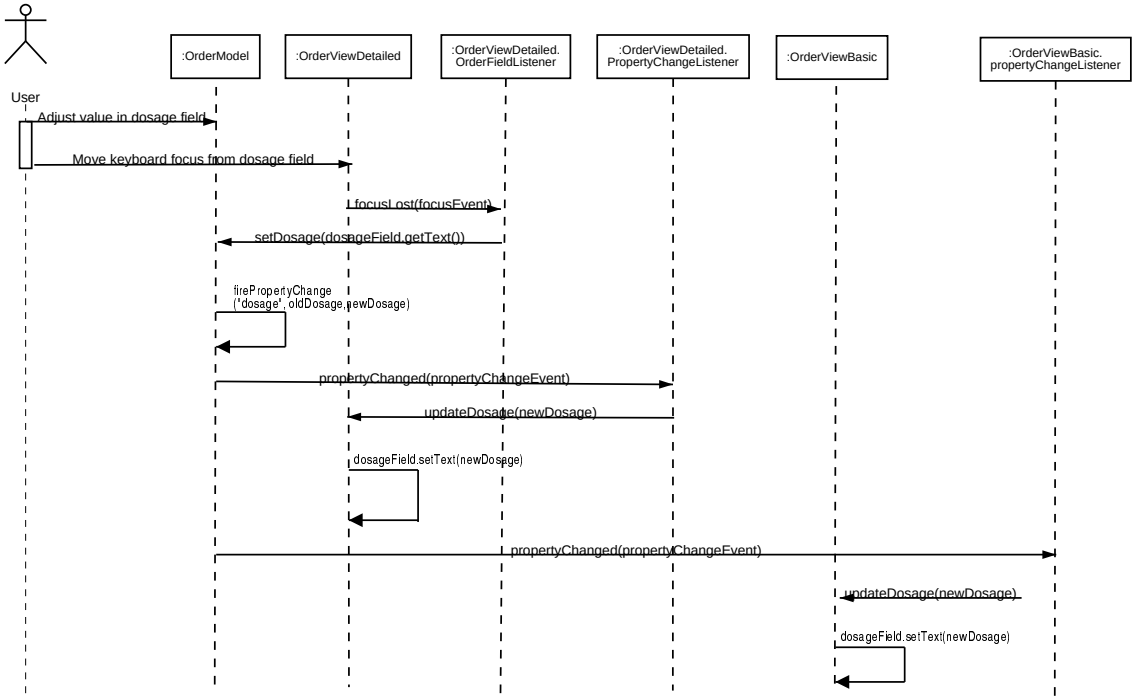


Figure 10.5. Prescribe design: Sequence diagram showing sequence for notification of Property changes in order model, following a user entering a new dosage value.

dosage and DrugModel's manufacturer. They may also be another Model, e.g. DrugModel is a property of OrderModel.

The Views relate to each other in the same way as the Models. PrescribeViewWorking contains a VectorViewCatalog showing a list of OrderViewBasics. When the user decides to edit an order, the OrderViewDetailed View is launched. This View contains a DrugViewSummary, some order parameters in text fields, and a VectorViewCatalog showing a list of AdminViewBasics. The user can also launch an AdminViewEditors from here.

10.3 The Multiple Model-View-Controller (MMVC) Framework

The MMVC framework consists of reusable software components, a source-code generator, and a pattern language demonstrating how it can be used. It was developed in conjunction with Prescribe, and also based on some ideas from MARCO. Apart from its presence in these products, it has also been applied with success in the Critique application discussed in Chapter 11.

The framework demonstrates how notions of usability can be embedded into tools for low-level reuse. The tools relate closely to the generic tasks outlined in Section 10.2.2. It must be re-emphasised that this work is proof-of-concept, since the framework has been developed mostly from the development of a single product. That is why only those generic tasks exemplified in Prescribe are considered.

In particular, the application shows how lists of objects can be supported in MVC. This is important to demonstrate reusability of design and code which supports the generic tasks Add, Remove, Enter Data (or Edit). The framework not only shows a programmer how to implement this functionality, but also dictates key features of the user-interface design and user-computer dialogue.

This is illustrated by the Vector Views in the Prescribe application. Each View shows a list of Models, and the user clicks on an object with the mouse to select it. When a list item is selected, its border is highlighted. When the user clicks on Edit or Remove, the action occurs on the selected list item. When the user clicks on New, a new item is created, selected, and able to be edited. These features, which pertain both to design and dialogue, are true for the Prescribe View (list of Orders), and the detailed Order View (list of Administrations). It would also be true of any other View within an application generated by the MMVC framework.

The level of guidance is of course not restricted to usability. The aim in developing MARCO, which is carried through here, was to produce a framework which supports traditional software engineering attributes as well as usability. This is a form of the detailed-design reuse within HCI that is a key objective of the thesis. In particular, MMVC is supposed to facilitate flexible, understandable, portable, code. These attributes are already provided somewhat by MVC, but are extended here with the introduction of the reusable components and the a treatment of the issues surrounding Vector classes. The framework, then, helps to demonstrate how tools for reuse can reconcile the tensions between usability and other software attributes which were articulated in Chapter 2.

10.3.1 Reusable Software Components

Prescribe introduces four reusable components. They are reusable in a broader sense than the reusability afforded by the autonomous nature of a Model and its Views. These are domain-independent components which could be reused in any application which follows the MMVC style, i.e. any application which is designed according to the patterns of Section 10.3.3.

10.3.1.1 Model and View

The first two components are Model and View, both interfaces rather than classes. Model declares the following methods:

```
public void addPropertyChangeListener(PropertyChangeListener newListener);
public void removePropertyChangeListener
    (PropertyChangeListener oldListener);
public void firePropertyChange(String propertyName, Object oldValue,
                                                    Object newValue);
public void refresh();
```

Model declares the methods necessary to notify listeners of change events, as was illustrated by Figure 10.5. It also contains a debugging method refresh(), which can be used to force an update in the event that an object's properties do not change. This is merely a device that can be used by the programmer to diagnose the cause of updating problems which may occur, and is not used in the final version of Prescribe.

The methods declared by View are:

```
public Model getModel();
public void setModel(Model _model);
public void removeModelListener();
public void updateAll();
```

Callers can set and get View's Model. Java's garbage-removal mechanism means that destructors are not present in the language, but it is still necessary to remove the View as a listener, hence an external caller can remove the View from the Model's list of listeners via removeModelListener. updateAll updates each Model-dependent control. This is used in setModel, so as to initialise each object according to the new Model. Occasionally it is used in other methods such as VectorViewCatalog.setComparator, which requires an update of the Vector View upon a request to set the new sorting mechanism.

Model and View contrast with the approach taken in MARCO. In MARCO, Models such as ClockModel extend Observable, which is effectively a proxy for Model. Likewise, Views extend a View class. In both cases, Java's lack of multiple inheritance implies delegation must be used to achieve the main functionality of the class. So ZoneModel, for instance, contains a TimeZone object which represents the current time, rather than being a subclass of TimeZone. Likewise, ZoneViewStatus contains a label to render the zone, rather than inheriting from the label class. In the present approach, classes implement Model and View methods, but are free to inherit from any class of their choosing. Thus, the Views in Prescribe directly inherit from the appropriate Swing components. Inheritance for Models is probably less important, since it could cause problems for the property change mechanism to take place (if a property were updated in a superclass created outside MMVC, how would the Model inform its Views?). However, Model has been left as an interface for consistency with View, and for future flexbility.

Multiple inheritance would still help, because there is behaviour implied in the Model and View interfaces which cannot be reused. As a compromise, a source-code generation tool has been developed, so the infrastructure code for Models and Views can at least be generated with relative ease. The opportunity was taken to create additional code which would also be used, e.g. constructors and test harness code. The tool is described in Section 10.3.2.

10.3.1.2 Vector Classes

The Model and View interfaces support some generic tasks, e.g. the concept of multiple Views facilitates the Compare task. But the major contribution of this framework is support for lists via the Vector components: VectorModel and VectorViewCatalog. VectorModel represents an ordered list of Models,

VectorViewCatalog is a View which presents the list to the user.

As discussed above, VectorModel consists of one property, a Vector object. The vector is used to store a list of Models. As would be expected for any MMVC property, an external caller can pass in a new vector via setModel and retrieve the Model via getModel. Listeners are notified when the vector changed.

An important lesson from MARCO which has been applied here is the pattern of Models listening to component Models. The vector is capable of listening to each individual Model, because it attaches a listener to the Model when it adds a new Model:

```
_model.addPropertyChangeListener(itemPropertyListener);
```

The effect is that the VectorModel will issue a property change notification whenever an item has changed. This behaviour is necessary for Views which do more than just render the Views of the individual items. In particular, the one example of a Vector View is VectorViewCatalog, and VectorViewCatalog requires this mechanism for sorting. Consider what happens when VectorViewCatalog is sorting AdminModels by the name of the staff member, and a user changes the staff member. Ordinarily, the AdminViewBasic inside the VectorViewCatalog would update itself, and VectorViewCatalog would be oblivious to the change that occurred. With observeItems set, the VectorModel will detect the change, inform VectorViewCatalog that the vector has changed, and VectorViewCatalog will then redraw the entire vector, so sorting will be preserved if the name changed.

Incidentally, the View would be redrawn if the user had only changed the actual time. The components are not specific enough to observe changes to particular properties of their contained components. Such functionality would improve efficiency, although it would make the components considerably more complex to use.

As a point of clarification, there will be considerable performance overhead from this approach, especially if it cascades to higher-level Models. Although this framework does not treat performance as its highest priority, there is still some consideration of each major issue. The conflict here is resolved in a relatively simple manner: VectorModel has an observeItems property, passed into its constructor, which dictates whether to perform this functionality. If the sorting capability is not used, and no other aspects of the View depend on the Views as a whole, the programmer can disable this mechanism. To this end, the code VectorModel uses to attach the item listener is surrounded by an inspection of the observeItems attribute:

```
if (observeItems) {
    _model.addPropertyChangeListener(itemPropertyListener);
}
```

VectorViewCatalog is the one View of VectorModel which has been created for this framework. This is a JPanel consisting of one View for each Model in the VectorModel. VectorViewCatalog is perhaps the most interesting class in the framework because it is very flexible. Its constructor is as follows:

```
public VectorViewCatalog(Class _knownModels[], Class _knownViews[],
```

```
                                    Comparator _comparator) {
```

The user passes in a list of Models and corresponding Views, from which a hashmap (modelViewClassMap) is created. This mapping, based on Java's reflection capabilities, enables any Model class to be associated with any View class. For example, some of the test methods use the following mapping:

```
Class knownModels[]={
    med.DrugModel.class,
    med.OrderModel.class,
    med.AdminModel.class,
};
Class knownViews[]={
    med.DrugViewSummary.class,
    med.OrderViewBasic.class,
    med.AdminViewBasic.class
};
```

This tells VectorViewCatalog that if it encounters any DrugModels inside the VectorModel, it must render them with DrugViewSummary. And similarly for the respective Order and Admin classes. How does VectorViewCatalog achieve this directive? The behaviour occurs in the updateVector method, which is called whenever the Vector changes. Firstly, an outer loop iterates through each Model in the new vector:

```
Iterator modelIterator=vectorModel.getVector().iterator();
while (modelIterator.hasNext()) {
```

An inner loop then iterates through the Model-View mapping to check if the Model can be rendered:

```
Iterator knownModelClassIterator=knownModelClassSet.iterator();
while (knownModelClassIterator.hasNext()) {
    Object knownModelClass=(Class) knownModelClassIterator.next();
    if (modelToView.getClass()==knownModelClass) { // A match has been found
```

Once the Model has been matched to a key in the map, the map is interrogated to obtain the corresponding View:

```
Class viewClass=(Class) modelViewClassMap.get(modelToView.getClass());
```

The View is finally ready to instantiate the View. It assumes the View is a subclass of JComponent, using an upcast to JComponent as a means of inserting the View onto itself (VectorViewCatalog is a subclass of JPanel, so it can add any JComponent onto itself). To instantiate the View, it must use Java's reflection API to instantiate the View and then set its Model.

```
// Create the new view as a JComponent
JComponent newView = (JComponent) viewClass.newInstance();


// Set the view's model to the current model
```

```
Class argClasses[]={Model.class};
Method setModelMethod=newView.getClass().getMethod("setModel", argClasses);
Object args[]={modelToView};
setModelMethod.invoke(newView, args);

// Add the view to the vector view (i.e. myself)
// and the convenience list of views
addView((View) newView);
validate();
```

The mechanism is very flexible. The Model-View mapping is set while the program is running, so a Vector can render any number of Model classes simultaneously (it would also be quite straightforward to create a public interface which changes the mapping after VectorViewCatalog creation time, so the vector could be requested to change the View it uses for a particular Model class). The simplicity was demonstrated by a simple experiment. A single line of PrescribeViewWorking, which created the View of the order vector, was altered. Initially, the mapping was:

```
Class knownModels[]={
    med.OrderModel.class
};
Class knownViews[]={
    med.OrderViewBasic.class
};
```

The View class for OrderModel was subsequently altered:

```
Class knownViews[]={
    med.OrderViewDetailed.class
};
```

The resulting View, admittedly a rather unattractive display, is shown in Figure 10.6. The basic order Views have been replaced by detailed Views. As the figure shows, the views can still be selected by clicking on them with the mouse (in a region where mouse-clicks have no effect on the detailed View). This can be used to launch an OrderViewDetailed View as in the regular application. Changes to the order arising either from the PrescribeView vector or the individual dialogs are reflected on the other Views corresponding to that OrderModel object, as would be expected.

A significant feature of VectorViewCatalog, briefly mentioned above, is that it can render more than type of Model simultaneously. There is no need to use this functionality in Prescribe, but a test method within VectorViewCatalog demonstrates how it can work with Prescribe components Figure 10.7.

Functionality of this nature would certainly be valuable in many applications. Indeed, the Swing component JList uses a Renderer object to achieve a similar result. In the case of VectorViewCatalog, the

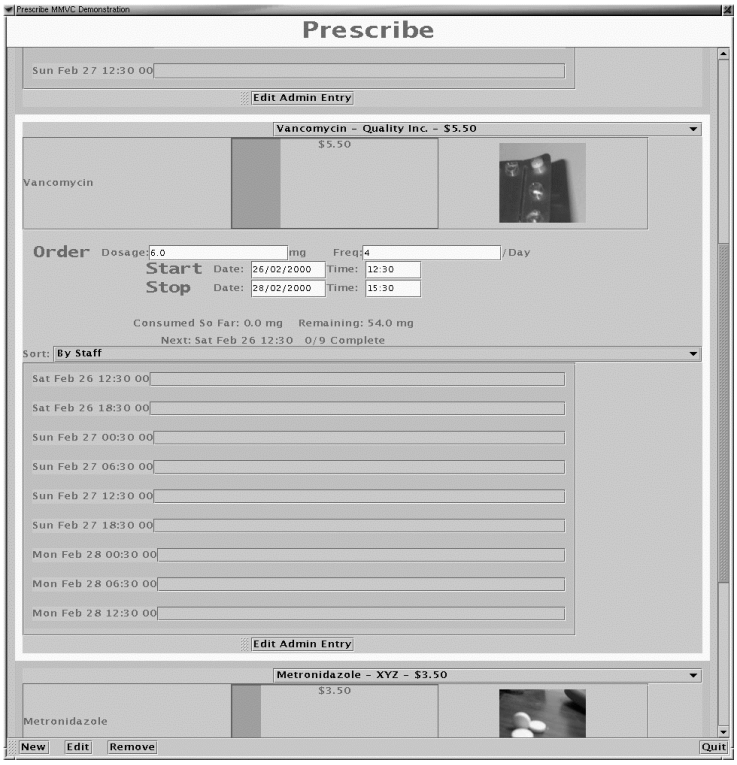


Figure 10.6. Prescribe testing screenshot: A main view of the application with OrderViewDetailed used to render Orders instead of OrderViewBasic.

functionality is implemented to relate specifically to the Model-View framework. The importance of this feature is demonstrated by considering what would happen if Prescribe were expanded to show not just orders but all treatments. Medical staff would sometimes want to trace the patients history by looking at a list of each treatment in chronological order. Although all Models would inherit from TreatmentModel, a generic TreatmentView would not be rich enough to capture the specific nature of the various treatments; from a user-centered perspective, the optimal way to represent the list would be a list consisting of Views such as OrderViewBasic, SurgeryViewBasic, and TestViewBasic. The reusable components here would make this straightforward - a developer would simply create the Models and Views, insert the Models into a VectorModel, and specify the Model-View mapping.

As well as rendering the View, VectorViewCatalog also facilitates user interaction by allowing the user to select a list item. When VectorViewCatalog adds a list item to itself, it registers a special listener class to catch mouse-button events. When the user clicks on the View, it highlights the item View's border in white (and unhighlights any previously-selected item). The View also produces a VectorSelectionEvent to notify registered listeners (which must implement VectorSelectionListener) that a selection has occurred. Multiple-item selection is not supported, but it would be straightforward to add this functionality using similar mechanisms. The selection mechanism facilitates several generic tasks within MVC. Most obviously, the Select task is supported. However, selection is useful for several other tasks too. Prescribe alone

Figure 10.7. Prescribe testing screenshot: VectorViewCatalog rendering a Vector with different types of models. A different view class is used to render each Model. From top to bottom, the Model classes being rendered are: AdminModel, DrugModel, OrderModel.

shows that it can help with adding and removing (Section 10.2.2). Of course, the reusable selection code itself does not indicate how these tasks should be supported; it only facilitates a few mechanisms for implementing the support. It is the role of the patterns (Section 10.3.3) to guide on using the code to support these tasks.

Finally, there is also a built-in sorting mechanism. At any time, the user can change the comparator which is used to sort the list. This functionality is necessary to let the user sort, a form of the Transform generic task. At present, the sorting is achieved indirectly via an insertion sort algorithm. It is indirect because it clears the list whenever the comparator changes, then inserts each View into its appropriate location (by first iterating along the partial list). In the future, the component could be made more efficient by performing a more efficient sorting algorithm upfront. This would have a more important benefit than enabling algorithms other than insertion sort to be employed; it would prevent the frequent change notifications which occur at present as the Views are shuffled through the vector during their insertion. This is, at present, the main bottleneck of the program. Notwithstanding this performance issue, the sorting mechanism works well, enabling the programmer to plug in new sorting methods simply by implementing a comparison function on the objects which are shown in the vector View.

10.3.2 Automatic Code Generation

There was a tendency for certain patterns to arise within the code of MARCO, leading to the idea that some source-code generation would be a useful complement to the patterns and components. MVCCreator [141] is a tool which was developed in parallel with Prescribe, a program which generates useful source code for Models and Views.

The user-interface (Figure 10.8) requests the user enter a package name, a root name for the Models and Views, all of the View's properties, and the View types. Complete examples of the source-code generated can be found in Appendix G. As an example, consider the constructor for OrderModel which can be created with the parameters shown in Figure 10.8:

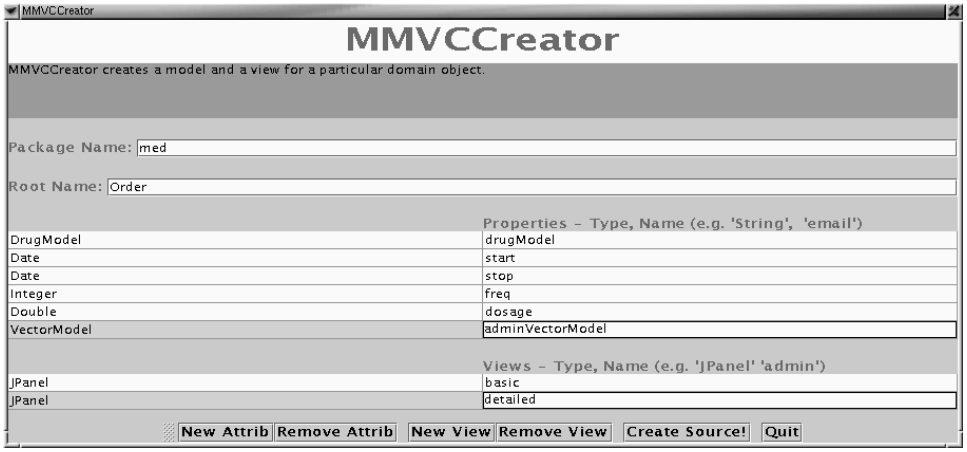


Figure 10.8. MMVCCreator screenshot: Generation of base source code for concrete subclasses of Model and View.

```
public OrderModel(DrugModel _drugModel, Date _start, Date _stop,
                  Integer _freq, Double _dosage,
```

```
                    VectorModel _adminVectorModel ) {
        setDrugModel(_drugModel);
        setStart(_start);
        setStop(_stop);
        setFreq(_freq);
        setDosage(_dosage);
        setAdminVectorModel(_adminVectorModel);
    }
```

The complete code output for the parameters in Figure 10.8 can be found in Appendix G.

MMVCCreator generates the following code for Models:

- Package declaration.
- Typical package imports.
- Class header.
- Declaration of PropertyChangeSupport attribute and attributes to store each property type. Some of these could be altered if the internal property were represented differently, but experience indicates that is a rare situation.
- Constructor, which sets property attributes.
- “get” and “set” methods for each property, which set the private attributes and fire property changes.
- The debugging method, refresh, which fires a property change.
- toString() method, which concatenates each property’s toString().
- Methods to allow external callers to add and remove property change listeners, and also to fire property changes.
- main() method, which creates a test object and outputs its toString() method.

MMVCCreator generates the following code for Views:

- Package declaration.
- Typical package imports.
- Class header.
- Declaration of private Model attribute, for the type of Model which is stored, and declaration of Model’s property listener.

- Constructor, which sets the Model and contains sample code for laying out controls and setting display options.
- getModel() which retrieves the Model, and setModel(Model), which removes the property listener from the existing Model, sets the Model, and forces an update (it also fires a property change, in case the programmer wants the View itself to be observed).
- ToString() method, which simply identifies its class.
- updateAll() method and update methods for each property of the Model. updateAll() calls each individual update method.
- PropertyChangeListener class, which calls the appropriate update method whenever the Model notifies a change has occurred.
- main() method, which creates a test Model and View, and places the View inside a test frame.

The code is a template only, and the View class in particular requires extensive additions and modifications. To this end, the View class contains many instructive "TODO" comments which the user should replace with appropriate code. The application has been helpful in creating the Models and Views in Prescribe, and also those in the Critique application (Chapter 11). In addition to supporting the MMVC patterns, it helps to produce consistent code and speeds development by providing useful methods for debugging, namely toString() and main(). Most of the generated code is in declarations and function headers. Some of the method bodies would be unnecessary if multiple inheritance were used, but most of this code requires significant tailoring.

MMVCCreator is by no means essential to the overall MMVC framework, since it simply speeds up coding and says nothing about usability or generic tasks. However, it does close the SE-HCI gap further, by reducing the effort required to move from specification to implemented system. Given that it occurs in the confines of a well-considered framework, this is an effective way to support prototyping. If the tool were developed further, it could allow for features such as round-trip engineering, in which the developer can manipulate either the code or the high-level specification. This would further improve responsiveness to the users. Developers would need to be wary of careless changes, but if handled effectively, a high quality of code would still be feasible.

10.3.3 MMVC Pattern Lanuage

The previous sections have discussed MMVC's reusable software components and source-code generator. These will not be effective in isolation; their usage needs to be documented. Moreover, they are too low-level to encapsulate siginificant knowledge about how to design for MVC in general, let alone how to support generic tasks in a usable manner. A pattern language can complement the components and generator by providing concrete advice on these issues.

The MMVC pattern language decribed in this section shows how even patterns of detailed software design can facilitate reuse of successful user-centered features, and to do so in a manner which incorporates reusable software components and software utilities. As such, it should be re-emphasised that the language is presented as a model of this approach, rather than a readily usable tool. Only MARCO and Prescribe are used as examples, whereas a pattern language should generally be based on a wide variety of examples. This is a slightly unusual type of pattern language, though, because its primary purpose is to document a software framework. If a developer of a software library were to follow this model and document their library via patterns, they too would probably be working from only one or two example systems. Therefore, although this is certainly a sub-optimal quantity of example systems, it is probably a realistic model for others to follow when documenting a framework.

The pattern language consists of nine inter-related patterns (Figure 10.9). The language starts with `MMVC Application`, which describes the application objects that the end-user runs, and the `Domain-UI Separation` discusses decomposition of the high-level Application objects.

`Domain-UI Separation` is the overall decomposition, and there is still a need to refine the architecture. `Vector` and `Vector Sort` deal with applications with list and guide on the usage of the reusable Vector classes. `Dependent Views` deals with the general Model-View relationship, and supports the reusable Model and View interfaces as well as the MMVCCreator tool. `Dependent Views` is followed by `View Placeholder`, `Model Comparison`, and `Model Summary`, all of which aid user-interface design. That is, they are patterns of useful user-interface design for the user, and also show how MMVC supports them. `Model Chain` helps programmers deal with the problem of a View which shows data aggregated from multiple Models.

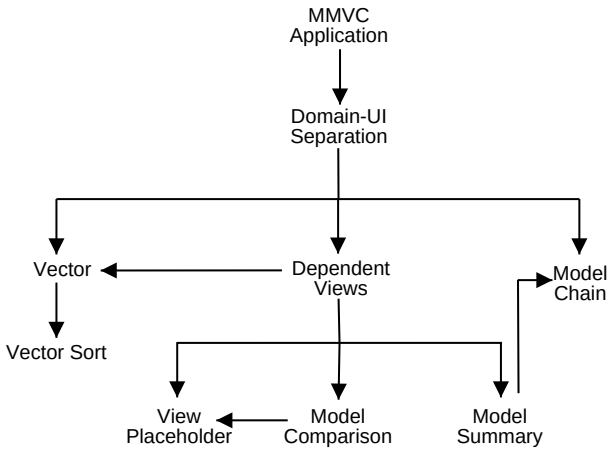


Figure 10.9. Map of MMVC pattern language.

Pattern 10-1. MMVC Application

Context: You have a high-level functional specification for your system.

Problem: How do you create the main application class in an MMVC application?

Forces:
- Your application has Models and Views.
- Each Model interacts with its Views. In addition, the Models interact with other Models and the Views interact with other Views.
- The main Application object which is executed by the user will have to ensure Models and Views are created and the user starts at the right place.

Solution: Create a single Model to represent the state of the application at any time. Create a View of the Application Model which the user will execute when starting the program. Since MMVC places Models in a hierarchy and Views in a separate hierarchy, the application is simply a View of the top Model.

As a variant, consider creating more than one View if there are different usage profiles (e.g. users with different needs or different usage contexts) and a single View cannot easily be parameterised to deal with the variation.

Examples: The PrescribeModel is a top-level Model which contains a vector of OrderModels, summarising the overall application state. PrescribeViewWorking is executed by the user and is therefore *the* application from an external perspective.

The MARCO application consists of a single Model, MarcoModel, which mediates between the ClockModel, LocationDBModel, and ZoneDBModel. The user has a few options for accessing MarcoModel. Typical users run MarcoViewFocus. Developers run MarcoViewStatus. People who only want the application in the background might run the demonstration View, MarcoViewBackground.

Resulting Context: You have a toplevel Model and its Views. Now decompose them using `Domain-UI Separation`.

Pattern 10-2. Domain-UI Separation

Context: You have designed the toplevel Model and its Views.

Problem: How do you create the overall architecture?

Forces:
- The architecture consists of Models and Views interacting with each other. There is a risk of haphazardly creating many associations, leading to high complexity.
- The architecture should capture the relationship among domain objects, as well as the relationship among user-interface objects.

Solution: Continue decomposing the Application Model until all Model classes are known and related to each other. Generate one or more View classes for each Model class, starting with the Application View. The Model decomposition will be somewhat parallel with the View relationship, e.g. if you have a Model M_1 which contains Models M_2 and M_3, Views of M_1 will likely contain Views of M_2 and M_3.

Examples: In Prescribe (Figure 10.4), there is a clear separation between the Models and the Views. The Models exist in their own design module and the same applies for Views. Of course, the association between Model and Views is inherited throughout.

A class diagram relating all Models in MARCO, centered around the Application Model (MarcoModel) is shown in Figure 9.3.

Related Patterns: Buschmann et al.'s Model-View-Controller pattern [34] also suggests starting by defining the Model, then designing Views.

The nested nature of Models and Views is an example of Gamma et al.'s [88] `Composite` pattern.

Resulting Context: Relate Views to their Models with `Dependent Views`. Ensure Models propogate their changes appropriately with `Model Chain`.

Pattern 10-3. Dependent Views

Context: You have breakdowns of Models and Views.

Problem: How do you relate Views to their Models?

Forces:
- The application should be capable of simultaneously showing a single Model under different Views, so that users can compare the Models and to account for individual variation among users.
- Avoid two-way dependencies between Models and Views. Views necessarily depend on Models, but Models should not depend on Views.
- With more than one simultaneous View, there needs to be a way to ensure the Views are synchronised.

Solution: When the View's Model is set, the View should store the Model, redisplay itself, and request the Model to send it property change notifications. At the same time, it should remove itself as a listener of any previous Models.

In MMVC, Models contain property change methods which utilise a set of listeners (implicitly via java.beans.PropertyChangeSupport). Whenever you change a Model's properties, you call firePropertyChange, identifying the property which changed. MMVCCreator provides template code for the Model's PropertyChangeListener.

The View should use a nested class, PropertyChangeListener, to listen to its Model's property changes. The PropertyChangeListener should know how to update the View according to the PropertyChangeEvents it receives. The suggested approach is for the View to define several update methods, to be called when different types of change events occur. In the simplest case, there will be one method for each property of the Model.

Related Patterns: The essential pattern here is the broadcast mechanism, described by Gamma et al.'s `Observer` [88] and Buschmann et al.'s `Publisher-Subscriber` [34].

Examples: All of the Models and Views in Prescribe work this way, as detailed in Section 10.2.4.

Resulting Context: Some useful components to place on your `Dependent Views` are ViewPlaceholder and ModelSummary (see `View Placeholder` and `Model Summary` patterns). Apply `Model Comparison`if you want to show several different Views of the same Model object, or several simultaneous Views of different Model objects. If your application has lists of Models, use `Vector`.

Pattern 10-4. Model Summary

Context: You are designing a View.

Problem: How do you make your View comprehensible to the User?

Forces:
- Some Views are complex, showing so much information that the user may have difficulty gaining a high-level picture of the situation.
- The user often does not have time to look at every detail.
- Often, you cannot represent all information about a Model on a single screen.

Solution: Include controls on your View which summarise one or more properties of the Model.

Examples: Prescribe's OrderViewDetailed shows dosage consumed so far and dosage remaining (see Figure 10.1). This is summary information, drawn from dosage-per-administration and administration vector. This View also shows information about number of complete administrations and next due administration, which is a summary of the administration vector. Likewise, the basic View summarises the order vector by showing if the order has an overdue administration.

Related Patterns: This pattern shows how to implement the `Abstract Mapping` safety-usability pattern (Section 7.4.1) under MMVC.

Resulting Context: If the summary depends not only on the View's Model object, but also on other Model objects maintained by this Model, you will probably require a `Model Chain`.

Pattern 10-5. Model Comparison

Context: You are designing a View which launches or contains other Views.

Problem: How do you relate the View to other Views?

Forces:
- The user should be able to show several View objects simultaneously. All of the following should be possible:
  - Work with Views of several different Model objects, e.g. two different items in a shopping catalogue.
  - Work with different View classes of the same Model, e.g. physical appearance and pricing statistics for an item in a catalogue.
  - Work with several different View objects, all from the same View class, of the same Model object, e.g. different portions of text for a single word-processing document.
- Some Views enable the Model to be altered.
- When there are multiple Views of the same Model, it is necessary to ensure they are synchronised, i.e. changes to one view are reflected in the other views.

Solution: Analyse user needs to determine when simultaneous Views are required. Then create these Views and use `Dependent Views` pattern to ensure they are synchronised. This is one of MVC's primary strengths — because all Views rely on the same data Model, it is easy to achieve by simply following `Dependent Views`.

Examples: PrescribeViewWorking allows the user to create as many OrderViewWorking Views as desired (Figure 10.3). These can represent different Models or different View objects showing the same Model object, the latter of which may be useful when the user wants to look at different sections of a large Administration list.

In MARCO, the user can access information about multiple locations simultaneously.

Resulting Context: You will need to consider how to structure the Views — perhaps they will be toplevel windows as in Prescribe, or maybe you will build a special View to contain them all. If you decide to contain the Views inside your main View, use `View Placeholder`.

Pattern 10-6. View Placeholder

Context: You are designing a View object which contains other Views inside it.

Problem: How do you place a View inside another View?

Forces:
- The *contained View* should be autonomous; i.e. it should track changes to its model itself, rather than relying on notifications from the *containing View*.

- You need to ensure the *contained View* will automatically update when its Model issues a property change notification.
- The entire Model object may be replaced by another Model object, i.e. the *containing View*'s Model actually removes the *contained Model* and inserts a new *contained Model*. When this happens, how will the *containing View* ensure the *contained View* is tracking the right Model object?

Solution: When a *containing View* contains a *contained View*, (semantically) detach the two Views via a *Placeholder*, i.e. the *containing View* holds a *Placeholder*, and the *Placeholder* holds one object: the *contained View*. The consequence is that when the *contained View*'s Model indicates its contained Model has changed, the placeholder can replace its *contained View* with a new View, constructed to track the new contained Model. With the placeholder occupying the same space as before, there is no layout issue to worry about.

Examples: In Prescribe, The DrugViewSummary object inside OrderViewWorking are contained inside a Java viewport, drugContainer. When OrderModel indicates its DrugModel has changed, the following code is issued (among other code):

```
if (drugViewSummary!=null)
    remove(drugViewSummary);
drugContainer.setViewportView(drugViewSummary);
```

The Vector View of the administrations works the same way as the Drug View.

In MARCO, MarcoViewFocus contains the location image inside a Panel object, imageContainer. When the location needs updating, the following code is issued:

```
imageContainer.removeAll(); // Should only be one panel removed.
imageContainer.add(theImagePanel);
```

Resulting Context: You have an autonomous View contained effectively inside another View.

Pattern 10-7. Vector

Context: You are creating the Models and Views in your application.

Problem: How can you handle ordered lists of Models?

Forces:
- Many applications have lists of domain objects.
- Lists in your application should be based on the Models and Views you are creating.
- There are several tasks users typically perform with lists: adding, removing, editing.

- A single application may have more than one list. The different lists may have different Views of items.

Solution: When there is a list of Models, use VectorModel and VectorViewCatalog. Consider using these classes to support adding new Models, and editing or removing existing Models. The Vector classes deal with the common problem of storing a vector of Models, rendering it according to the users needs, and allowing the user to manipulate it. You use them as regular objects: set the vector property of VectorModel with a vector containing any number of Models, of heterogeneous types if desired. Specify the Views which will be used to show each Model class, the comparator (see `Vector Sort`), and create a View which can be used like any other View.

If the vector is editable by the user, you may wish to produce a summary vector, which lets the user launch individual edit dialogs. Follow these guidelines:

- When the user wants to **edit** an existing element, produce a dialog View of the selected item. Interrogate the Vector View to determine the selected item.
- When the user wants to **create** a new element, add a default Model to VectorModel's vector, select it for the user, and allow the user to edit it via the same dialog View as for editing.
- When the user wants to **remove** an existing element, determine the selected View, interrogate it to determine its Model, and remove the Model from the VectorModel's vector property.

On the other hand, if the vector manages small Models, you may wish to contain small editable Views inside the vector.

Examples: PrescribeModel contains a VectorModel of OrderModels. PrescribeViewWorking renders the Vector by containing a View of type VectorViewSummary. The three operations above are all supported. The classes' capability to enable direct editing of editable Views was demonstrated in Figure 10.6.

Resulting Context: You can represent Vectors. Allow the user to sort them with `Vector Sort`.

Pattern 10-8. Vector Sort

Context: You have a Vector arrangement.

Problem: How can the user sort the vector?

Forces:
- Users like to sort lists to help them find information more quickly and spot overall trends.
- The user may have different Views of a Model open at the same time, each with a different sorting method.
- As requirements change, you need to easily introduce and remove sorting methods.

Solution: When the user requires lists to be able to be sorted in different ways, create Comparators to reflect the user's expectations, and pass them to VectorViewSummary.

Examples: The administration list in Prescribe's OrderViewWorking has several useful sorting methods: sort by name, sort by time of administration, sort by discrepancy between time of administration and expected time. A combobox lets the user change sort order. When the combobox is pressed, a callback method maps combobox values to Comparators, and passes the correct comparator to the VectorViewSummary. For instance:

```
if (order.equals(STAFF_SORT)) {
    adminComparator = new AdminComparatorStaff();
}

....

adminVectorView.setComparator(adminComparator);
```

The Comparator subclasses simply contain a compare method, for instance (AdminComparatorStaff):

```
public int compare(Object _adminModel1, Object _adminModel2) {
    return ((AdminModel) _adminModel1).getStaff().compareTo(
                    ((AdminModel) _adminModel2).getStaff());

}
```

Related Patterns: The use of Comparator is an example of Gamma et al.'s `Strategy` pattern [88].

Resulting Context: The user can sort the Vector.

Pattern 10-9. Model Chain

Context: You are designing the relationship among domain Models.

Problem: How can you ensure Views reflect the current state of Models?

Forces:

- Models may change at any time, from a source other than the View you are designing.
- In MMVC, a single View can only listen to one Model.
- A Model can be associated with other Models, and the View may reflect the state of these Models. How will the View know when these associated Models are updated?

Solution: When a View reflects a Model that is associated with other Models, request the View's Model to listen to the other Models. This is used when a View does more than just inserting Views of Models. This would occur, for instance, if the View's Model contained two image Models, and the View showed a merged image. If either image Model changes, the merged image will need to be updated. This can only occur if the View's Model triggers a change notification.

This pattern must be used with caution, because a large chain of Models can be inefficient. Thus, use it only when necessary, rather than routinely applying it to any Model which contains another Model.

An alternative to this pattern is for Views to directly register with other Models. This will be more efficient in some situations, but there is a danger of making the framework too complex, with change notifications being difficult to follow. It would also entail significant code overhead if each View contained a separate handler for each observed Model.

Examples: PrescribeViewWorking contains a VectorViewSummary of AdminViewBasic Views. If the user alters an AdminViewBasic, how will sorting order be preserved, since only the individual AdminModel will be trigger a notification? The solution has been to apply this pattern, i.e. The VectorModel observes its constituent Models. The Vector Model provides support for this capability via its constructor argument, observeItems.

In MARCO, LocationDBModel observes each LocationModel. Thus the Location View can show a list directly from LocationDBModel, i.e. without having to plug in numerous single specialised LocationView items. If the user changed the name of a LocationModel, the change would propogate to LocationDBView.

Resulting Context: The View receives all relevant notifications and can therefore keep synchronised with all relevant Models.

10.4 Discussion

The MMVC framework shows how detailed design tools can impact on usability. It shows how the tension between usability and other software attributes can be resolved, within a specific user-interface architectural style.

The framework and sample application were developed according to several generic tasks. It was expected that the patterns, once developed from the generic tasks, would be quite independent from them. This is because the generic tasks in themselves are neither software design patterns nor usability patterns in the style assumed throughout this thesis, even though they are "patterns" of tasks supported in software. They cannot be called usability patterns because they are too simplistic and in the form presented in Chap-

ter 8, bear no relationship to each other (this would not be true for Breedvelt-Schouten et al.'s [30] task patterns).

There are several examples of patterns in this style. Most notably, `Vector` represents an amalgamation of needs implied by Add, Edit, Select, Enter Data, and this is also true for the reusable components VectorModel and VectorViewSummary. In a similarly vein, `Domain-UI Separation` and `Dependent Views`, which represent the basic MVC concepts, implicitly support Compare by setting the groundwork for two or more synchronised Views. The fact that patterns could show how to implement generic tasks without having to be closely based on them demonstrates the generic nature of a pattern language. The user of a pattern language implicitly creates a system which supports certain principles without having to be explicitly aware of them. Gabriel's tennis metaphor (Section 3.2.2) is pertinent: a player can learn to hit the ball smoothly and powerfully by being instructed to aim for a point just beyond the ball, rather than simply being told to hit the ball smoothly and powerfully.

Nevertheless, it proved to be useful to include patterns which did in fact map closely to particular tasks. `Model Comparison` carries the aforementioned patterns further by emphasising that multiple Views are possible. The patterns' forces are user-centered goals combined with practical engineering considerations. The solution explains how to create a comparison in programming terms, i.e. multiple Views for one Model. In the same way, `Model Summary` is closely related to the Summarise generic task. Once again, the solution is pitched at the programmer — "Include controls on your View which summarise one or more properties of the Model". The second aim of this thesis is to reuse detailed software design features that improve usability. A solution like this shows how this objective can be achieved.

The close mapping of some patterns to generic tasks, though initially not expected to be a useful concept, turned out to be quite helpful. Because unlike the generic tasks themselves, the corresponding patterns do involve recurring relationships within software design. They contain insights which would not be immediately apparent to a designer, and are closely related to other elements in the framework, i.e. other patterns, reusable classes, and the source-code generation tool.

The inclusion of the reusable classes and source-code generation tool helped to produce a more coherent framework than would be possible with the patterns alone. The Vector component is able to encapsulate certain decisions regarding the user-interface and user-computer dialogue, the way selection occurs and is shown. However, this alone is not enough to ensure the component is used as the programmer intended. A programmer could implement the "Edit" task in several ways, e.g. select then click on remove button, double-click, right-click to reveal popup menu, select then enter keyboard shortcut. The component is flexible enough to do any or all of these. Instead of expecting the programmer to decide, the `Vector` and `Vector Sort` patterns show what was intended. This improves human factors consideration and ensures different vectors in the program will support the same control mechanisms, i.e. contributes to a consistent feel.

A framework of this nature can in fact facilitate consistency at a higher level than the usage of individual

components. The example of a pattern language for the Apple Macintosh discussed in Figure 3.2.3.4 is illustrative of the type of role a large-scale version of MMVC could play. If the `Vector` pattern had suggested that users produce a popup menu in order to edit an item, then one would expect other patterns to also suggest popup menus. In other words, the patterns would show how certain themes or principles are carried out across the entire library. In the case of the Macintosh, patterns would show how to implement drag-and-drop for various scenarios.

Like Planet and Safety-Usability Patterns, MMVC's pattern language is tightly-constrained. In this case, it applies to systems with several domain Models, a user-interface, probably some manipulation of items in lists, and probably a need for multiple Views of the same domain Models. Furthermore, it cannot be used effectively without the reusable components and source-code generator. The point of this extent of specificity is to produce as coherent a framework as possible. While the framework could not be used for all software, it could certainly form a model for an organisation to create its own framework for internal purposes. After all, an individual organisation is likely to develop applications with (a) similar architectural style, (b) the same reusable components, and (c) tasks typically performed in a certain domain, as was discussed in Section 8.7. A combination of reusable components can be developed, some domain-specific (like MARCO's LocationModel and Views), and some domain-independent (like VectorModel). Then, the pattern language forms a medium for teaching people how to design with the components and for ongoing improvement.

It is not difficult to see how MMVC could be expanded. There are many generic tasks which are not directly supported by the framework. A particularly interesting group is the meta-tasks: Preview Task Effect, Obtain Guidance, Undo, Update System State, Change Task Parameters. These tasks could lead to a thread of patterns, perhaps based on the concept of a "Task" class, similar to Gamma et al.'s `Command` pattern. `Command` shows how a command can be used to undo and redo. In MMVC, Undo and other meta-tasks would be described in the context of the overall framework. Perhaps a task would be represented as a Model. Then, a View of the task could be used to show what will be undone if the user elects to Undo (e.g. "Undo Image Creation"). A history list could be implemented with the same task View (i.e. "Image Creation" would be shown on the history list). Similarly, Obtain Guidance might require a different task View to implement a Help interface. Perhaps task Views for help would provide methods like getExplanationText(), showExample(), and so on, which would enable them to be plugged into a Help window which presents the help in a consistent manner. Expanding the framework in this way might lead to a library and supporting pattern language which is considerably more broadly applicable than MMVC.

MMVC contains patterns of detailed software design. It shows how tension between certain software attributes, especially flexibility and understandability, can be reconciled with usability concerns. The pilot study of Chapter 8 provides some evidence that people can design effectively with generic tasks. The brainstorming capability of generic tasks combined with the detailed design support offered by MMVC suggests a cohesive development process which provides good support for incremental development. As

shown in Figure 8.2, development begins with identification of generic tasks, along with other activities, and proceeds with the aid of a framework like MMVC. If a future change involves addition of a new generic task, the MMVC framework should assist development, because the existing software is already built with the framework.

For example, an office application might show a list of departments. Under MMVC, this would be implemented using a VectorModel of DepartmentModels, and a corresponding VectorView and DepartmentView. As the organisation expands, a user may request the list be sorted alphabetically. Because the initial system uses the Vector classes, it will be relatively easy to support this change request, by creating a new subclass of the Comparator class. The `Vector Sort` pattern will provide guidance. Later still, requests might come to create new sorting orders and let the user choose the order. Again, the framework will support these high-level tasks. Likewise, the framework will allow a read-only list to become a menu supporting selection. All of this is achieved by a combination of the reusable software components and the instruction provided by the patterns.

However easily the tasks can be mapped to software patterns, there is still an opportunity to take the concept a step further, and use patterns throughout the process. Planet and Safety-Usability patterns have already demonstrated that high-level patterns can support usability, supporting the first aim of this thesis. MMVC demonstrates that low-level patterns can also address usability, supporting the second aim. As a combination of high-level and low-level reuse, the generic tasks combine with MMVC in a manner which relates somewhat to the third aim. However, this aim of combining the approaches can be addressed more directly by considering how a single pattern language can be used for both high-level and detailed design. In the next chapter, the high-level patterns of Planet are augmented with a set of software design patterns at a similar level of abstraction to MMVC patterns.

Chapter 11

Weaving High-Level and Low-Level Patterns: An Extended Version of Planet

11.1 Introduction

Throughout this thesis, there has been an emphasis on the *language* aspect of pattern languages. As presented in Chapter 5, the Planet language contains patterns for both organisational process and high-level design. It has been argued that Planet is a language because the patterns share a common goal and collaborate in a coherent manner. By focusing on the specialised requirement of catering for culturally-heterogeneous users, a tight set of relationships among patterns was achieved. This chapter takes an important step further, adding detailed software design patterns into the mix.

The previous chapter has explored the notion of detailed software patterns which improve usability. That work, however useful for software designers, does not capitalise on the capability to combine patterns at different levels of abstraction. Planet's high-level design patterns are ideally suited to act as a springboard for detailed design patterns. They represent a specific vision about the functionality and user-interface. They also explain how the user can configure their settings. The extension of Planet contains patterns describing the implementation of these features. It directly addresses the third aim of the thesis, which involved using a single knowledge base to map from high-level, usability-motivated, design knowledge to detailed design knowledge.

Before describing the detailed Planet patterns, an example application will be discussed in Section 11.2. This application, Critique, was developed to serve as an example of the Planet patterns. Its user-interface, functionality, and feature configuration follow the concepts described in Planet's high-level design patterns.

Its software adheres to the detailed design patterns of Planet. The patterns themselves use Critique as an example, as well as seeking examples from other applications, and are documented in Section 11.3. The discussion in Section 11.4 indicates how the complete version of Planet consolidates many of the concepts introduced throughout the thesis.

11.2 Example Application: Critique

11.2.1 Walkthrough

Critique [140] is a small program which demonstrates the application of many Planet patterns. The idea is inspired by Nielsen's observations on a French educational product [171]. The product allowed the teacher to annotate the poems, and Nielsen suggested that, in some other countries, permission would be granted for students to annotate the poems. Critique assumes that both students and teachers have rated a photograph according to some criteria. It shows the photograph, the ratings, and an overall score. In the version which is nominally termed "English", a table shows all criteria for all raters, i.e. teachers (top row) and students (Figure 11.1). In the "French" version, only the teacher's ratings are displayed (Figure 11.2). A score is also shown in both versions. For the English version, this is calculated as the average of all ratings across all raters. For the French version, it is an average only of the teacher's ratings.

The application is controlled by a number of preferences (Figure 11.3):

Background Colour Major background colour for the overall user-interface.

Font Font used for welcome message and score.

Language Natural language used for all text[1].

Evaluation Style Whether or not student ratings count. This choice determines both (a) whether student ratings are shown, and (b) whether student ratings influence the overall score.

Number Format Number format to display overall score; may be zero, one, or two decimal places[2].

The choice of cultures here is rather arbitrary. Nielsen's example is only speculation, and more research would be required, along the lines of Planet's organisational patterns, to determine exactly what functionality is required. The cultures are simply used as examples for this problem. Another point about the application is that the rating data is read-only. It nevertheless serves to demonstrates the high-level software patterns and detailed software patterns which are essential to internationalisation.

[1] For ease-of-implementation, the preferences dialog is displayed only in English. It would be possible to store labels for the Preferences dialogue within the locale-specific MessageBundles, but this was deemed unnecessary for this proof-of-concept application.

[2] This is implemented with a standard NumberFormat class in Java, hence could easily be used to provide formats such as X,XX or XX%

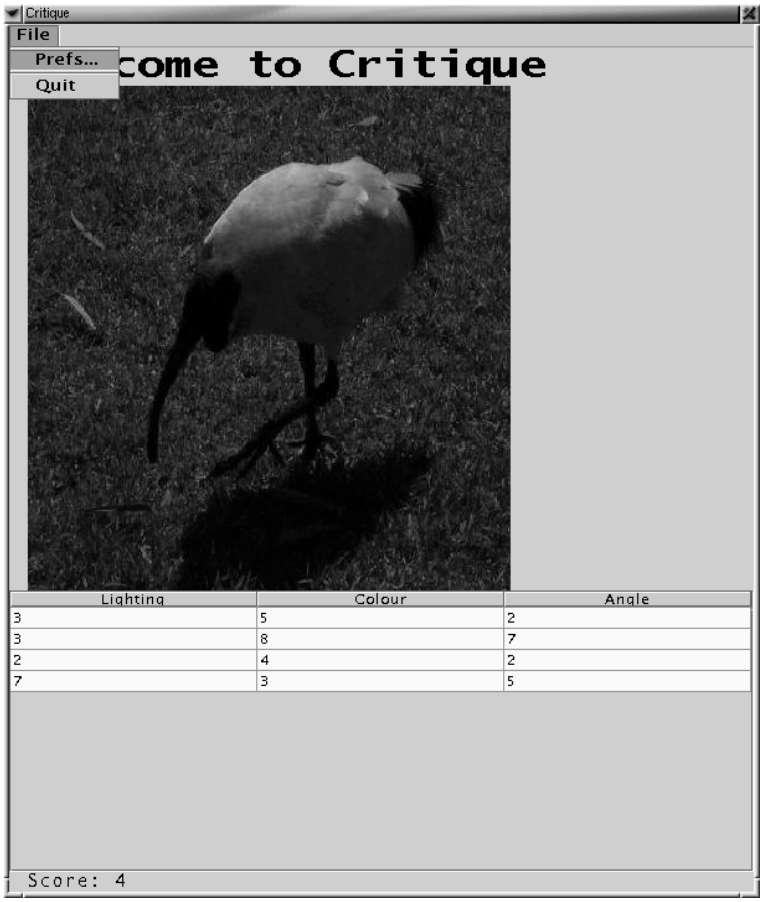


Figure 11.1. Critique screenshot: Default appearance for "English" culture. Following "English" defaults, the language is English and evaluation style is "Everyone" (which affects the overall view layout and the calculation of score as the average across all attributes for all raters). Following application defaults (for which no overriding English settings exist) , background color is blue, font is Courier, and number format is zero decimal places).

A critical aspect of the preferences is the flexibility involved. While the software has been based on "English" and "French" users, the preferences in fact allow a wide variety of choices. Each of the five variables can be set as desired. Thus, a user can receive the so-called French teach-only evaluation style, while looking at the text in English and viewing a background colour, font, and number format which is not default for either French or English (see Figure 11.4).

If the user can have any settings, what relevance does culture have? Firstly, an analysis of culture still drives the creation of some combination of variables in the first place — if there were no cultural basis to evaluation style, it may not have been useful to make the style flexible. The second result of cultural consideration is the one which users will directly interact with. This is the notion of inclusion of default values espoused by the Planet's `Cultural Profile`. There is a universal default for each variable. Culture profiles may override the default, but this is optional. The universal default for language is English, which

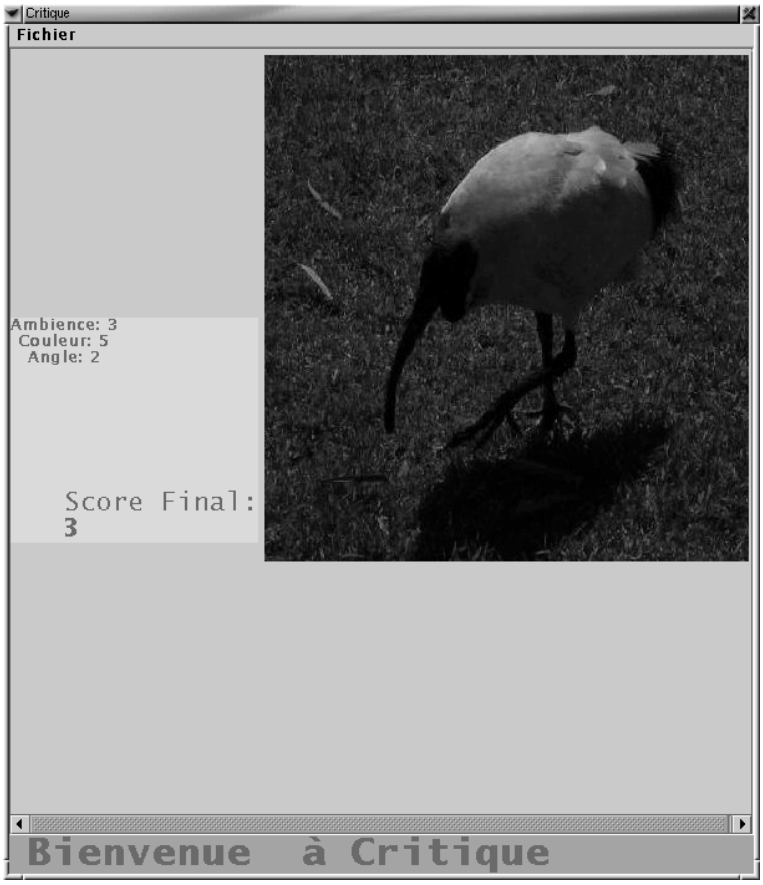


Figure 11.2. Critique screenshot: Default appearance for "French" culture. Following "French" defaults, the language is French, evaluation style is "Teacher" (which affects the overall view layout and the calculation of score as only average of only the teacher's marks), and the background colour is aqua. Following application defaults (for which no overriding French settings exist) , font is Courier and number format is three decimal places).

the French profile overrides. The universal default for font is Courier, which the French profile does not override. Therefore, when a user resets the preferences dialog to "French", the language becomes French, because the French preferences specify this variable, while the font becomes Courier, because the French preferences say nothing abou this variable.

11.2.2 Implementation of High-Level Design Patterns

At some level, Critique demonstrates each of the high-level design patterns in Planet (Figure 11.5). Because it was developed as a basic prototype rather than through a detailed analysis process, it does not demonstrate the application of the organisational patterns. However, the link between these two levels should be clear; the decisions regarding culture which are inherent in Critique could realistically derive from the repository-centered analysis process.

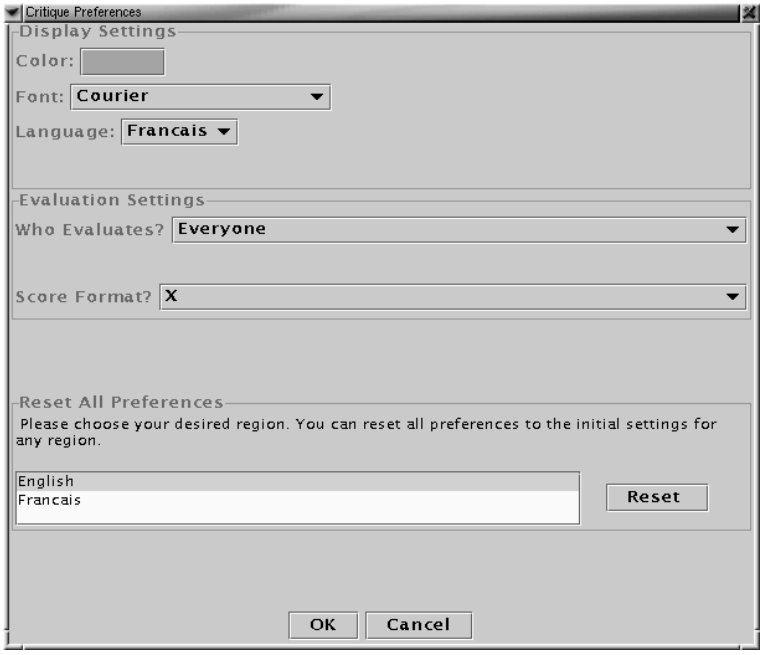


Figure 11.3. Critique screenshot: Preferences Dialog. User can change settings related to display and evaluation. The culture-dependent settings are integrated into the dialog in a natural way, rather than having a separate box for "cultural factors". The user can also reset preferences to the defaults for a culture. This causes the values on the dialog to update. The settings in the dialog correspond to the view in Figure 11.4.

Later in this chapter, the detailed design of Critique is used as an example for several detailed-design patterns. It is important to establish that this detailed design applies to a system which follows the *high-level design* guidance of Planet. The link shows that the high-level patterns can lead to a high-level design whose detailed design can be implemented according to the low-level patterns. This capability is central to aims of this thesis, i.e. reusing knowledge about high-level design and its mapping to detailed software design. With the full version of Planet, it is possible to work with a set of patterns to create a high-level design, then work with related patterns to implement the design.

Here, then, is an explanation of each design pattern and its relation to Critique.

`Elastic User-Interface` The layout of the teacher view and the evaluation view are completely independent of each other.

`Flexible Function` The score is calculated according to evaluation style: teacher only or all raters.

`Targeted Element` The language, background colour, font, and number format, can all be customised.

`Universal Default` The application provides fallback settings for the case when the culture is not explicitly supported, i.e. is neither "English" nor "French". If desired, a new culture could be added and it would initially take on universal defaults. Preferences could be tailored over time.

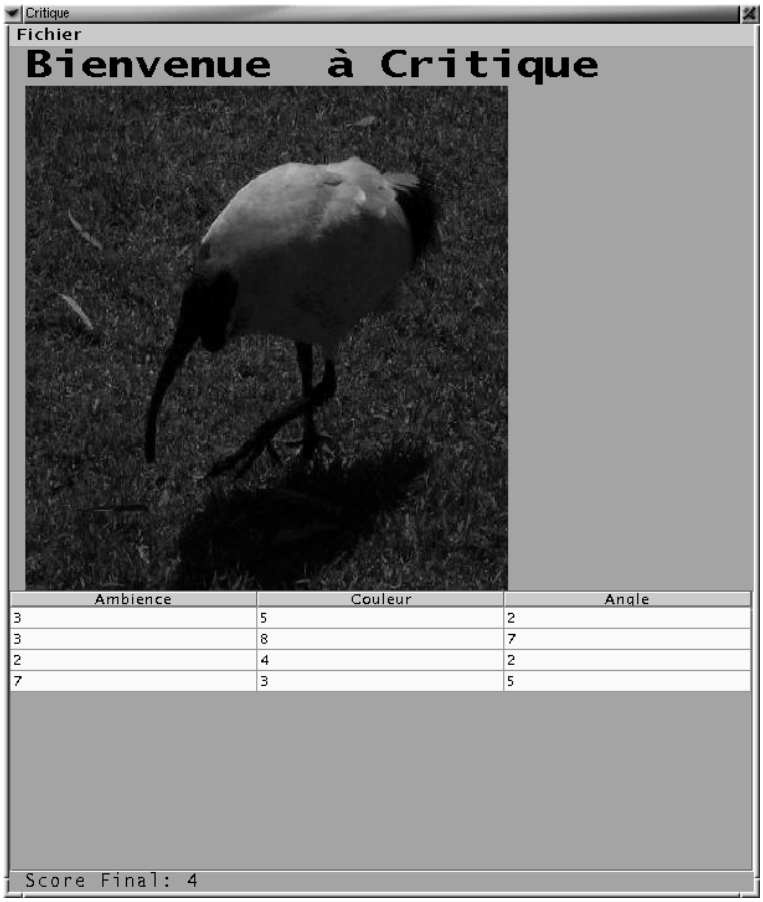


Figure 11.4. Critique screenshot: French language with "Everyone" evaluation style. Since "Everyone" evaluation style is not the default French setting, this view illustrates how Critique enables settings tailored to the individual user's needs. The view is generated by the settings shown in Figure 11.3.

`Cultural Profile` Cultural profiles are provided for English and French; the user can set all preferences according to their desired culture with the `Reset` button on the preferences dialog.

`Citizen ID` The application uses Java's `Locale.getDefault()` method to guess the user's culture. This is used to set the initially-selected value of the culture profile listbox on the preferences dialog (the profile is not activated until the user actually hits the Reset button).

`Integrated Preferences` Preferences are organised according to their individual nature, rather than on the basis of their culture-specificity. To wit, language is placed with the other display settings — colour and font. The culture-specific "Who Evaluates?" is shown in the same box as the culture-neutral number format; both relate to evaluation.

`Preference Group` Language is a preference group — selection of French leads to numerous string resources being set to French. "Who Evaluates?" is also a preference group. If set to Teacher Only,

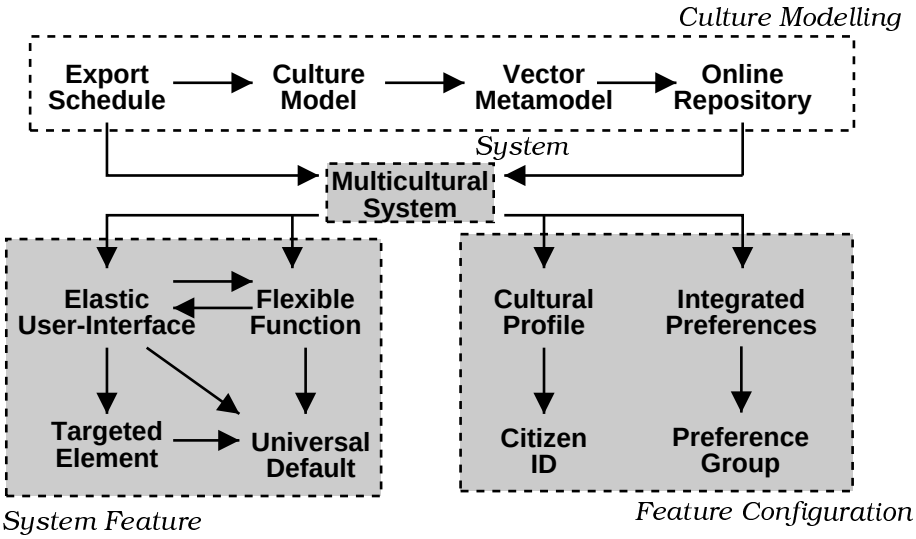


Figure 11.5. High-level patterns in Planet. The high-level design patterns are shown in grey —- these are the patterns demonstrated in the Critique application.

the user receives the Teacher view and the score is calculated by teacher. If set to Everyone, the user receives the Everyone view and the score is calculated for everyone. Therefore, setting "Who Evaluates?" causes both evaluation algorithm and view type to be set.

11.2.3 Software Design

11.2.3.1 Static Model

A sketch of Critique's architecture is shown in Figure 11.6. The ArtModel class represents the image and ratings, and can be rendered by both ArtViewEveryone and ArtViewTeacher. Critique creates a toplevel frame which can store one of the Art Views. For ease of implementation and understandability, there is no ArtView class from which ArtViewEveryone and ArtViewTeacher inherit. There is certainly a reasonable degree of commonality among these classes and it would be worthwhile creating such a class if high maintainability were required. Critique itself creates a frame, an ArtModel, and inserts the appropriate view.

Critique holds the single object preferenceBundle, which defines all preferences, whether or not they are culture-specific. The preferences are stored using java.util.ResourceBundle. This class is especially for internationalisation, enabling the programmer to declare several objects which are culture-specific. Culture-specific resource-bundles provide values for the objects. The ResourceBundle is used hierarchically, i.e. the overall preferenceBundle object contains a messageBundle defining language-specific strings and an evaluationBundle defining which ArtView is used and which scoring algorithm is used.

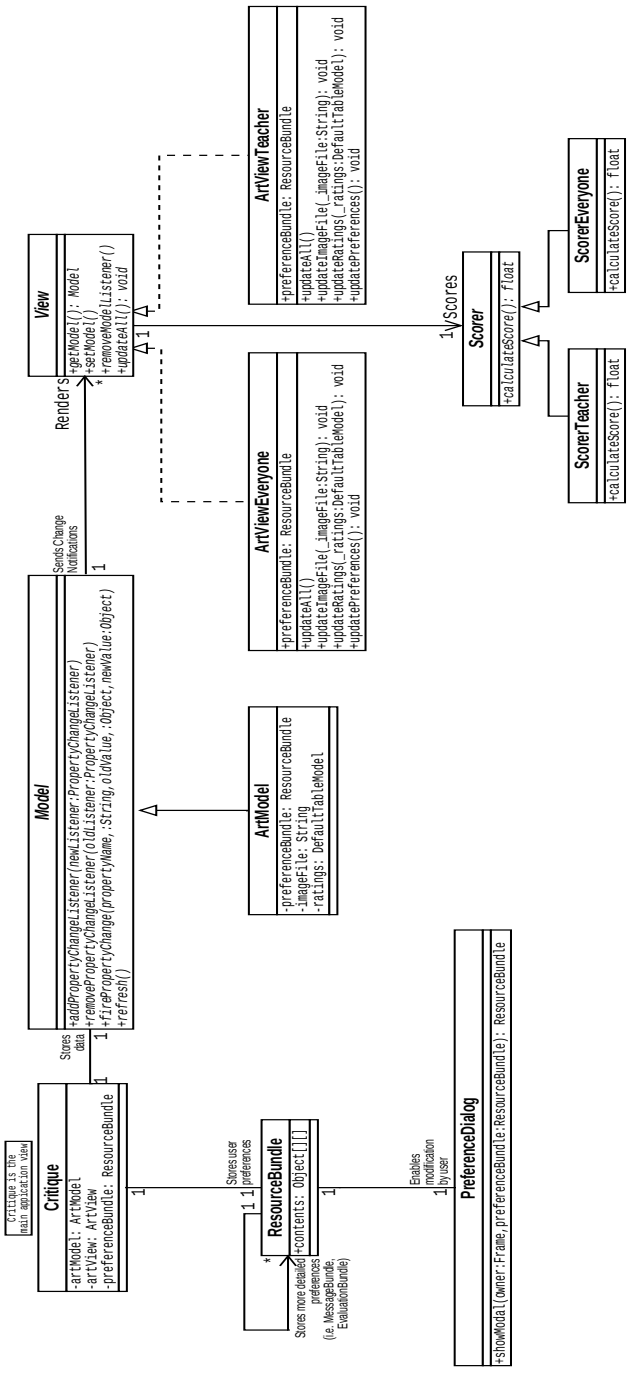


Figure 11.6. Critique design: Static Class Diagram in UML notation.

The scoring algorithms themselves are an example of the `Strategy` pattern [88]. This pattern requires a superclass to declare an abstract method, and extending classes to implement the method. A client class can then "plug in" a strategy by selecting the class which defines the desired strategy. Here, Scorer declares the abstract method `calculateScore`. The preferenceBundle object contains a reference to one of Scorer's subclasses. When the view renders the score, it looks up the current concrete Scorer object, and calls its

`calculatescore` method.

On a different note, the Model-View separation is achieved using the MMVC framework (Chapter 10); ArtModel and its corresponding views were initially generated with the MVCCreator tool, they inherit from the standard Model and View classes, and they interact according to the MMVC design patterns. In this respect, the application provides a further application of the core MMVC concepts.

11.2.3.2 Handling Preferences

This section elaborates on the central feature of Critique, the preferenceBundleobject, and its relation to the default culture profiles. Several code samples are offered.

Early in the Critique constructor, the locale is set:

```
Locale currentLocale=Locale.FRENCH;
```

Incidentally, Locale.getDefault() could have been used instead to obtain the default locale in the user's virtual machine, just as PreferenceDialog uses it to set the default culture to reset to (Section 11.2.2). However, it was decided to hard-code the default to make the program to facilitate analysis.

currentLocale can then be used to obtain the initial value for preferenceBundle:

```
preferenceBundle = ResourceBundle.
                   getBundle("critique.PreferenceBundle",currentLocale);
```

With this preferenceBundle, it is possible to obtain culture-specific objects. For example, the default font for French-culture can be obtained as follows:

```
Font frenchFont = preferenceBundle.getObject("FontFamily"),Font.BOLD,30));
```

The preferenceBundle, an instance of ResourceBundle itself contains two ResourceBundles: messageBundle and EvaluationBundle. These are extracted into convenience member variables before being used. For example:

```
ResourceBundle messageBundle = (ResourceBundle)
                    preferenceBundle.getObject("MessageBundle");


...  (later)    ...


String scoreMessage = (String) messageBundle.getObject("Score");
```

Since the secondary bundles are encapsulated in the preferenceBundle, only the preferenceBundle is passed between modules.

The evaluationBundle is particularly interesting because it contains two major resources: the View

object and the Scorer object. These are obtained in the same way as messages, i.e. [3]

```
ResourceBundle evaluationBundle = (ResourceBundle)
                    preferenceBundle.getObject("EvaluationBundle");

JComponent artView = (JComponent) evaluationBundle.getObject("ArtView");
getContentPane().add(artView,BorderLayout.CENTER);

Scorer scorer = (Scorer) evaluationBundle.getObject("ScorerStrategy");
float score  = scorer.calculateScore(_ratings);
```

The main Critique object creates a menu which enables the user to open the preferences dialog. When Critique receives a new preferenceBundle from the dialog, it first updates its own appearance. This amounts to providing correct translations for its menu. Since there is substantial overhead in changing the view, Critique performs an optimisation: it checks if the new preference requires a view change using Java's reflection abilities:

```
Class currentViewClass=artView.getClass();
Class desiredViewClass=
    (evaluationBundle.getObject("ArtView")).getClass();
if (currentViewClass != desiredViewClass) {
    // Remove current view and insert new view
}
```

The broadcasting mechanism already present in the models and views has been exploited allow broadcasting of preference changes. Thus, after setting the view if necessary, Critique only calls artModel.setPreferenceBundle. The art Views will notice the change and update themselves.

11.3 Extending Planet with Detailed Design Patterns

11.3.1 Development of the Pattern Language

The detailed design patterns were initially developed by analysing several internationalisation-based programs, and by speculating on useful design concepts for implementing the high-level Planet patterns. Critique was created to verify these patterns and it also helped to refine them.

In the Chapter 7 study on safety-usability patterns, a number of guidelines for pattern authors were offered in accordance with observations. The detailed Planet patterns were created after the safety-usability study was conducted and the guidelines were followed for two reasons. First, while it was mentioned the

[3] This is a valid sequence, but one that does not exist directly in the program; it is present across several code segments in the program.

guidelines should not be considered conclusive, it was felt that they still contained enough insights to benefit the documentation of a new set of patterns. Second, the detailed Planet patterns provided a suitable way to demonstrate implications of the guidelines. Following are the guidelines again, along with their influence on the patterns introduced in this chapter:

Demonstrate with Good Examples Each pattern has at least two well-explained examples. The Critique application serves as a common reference point for each pattern.

Provide Summary Views The next section (Section 11.3.2) contains a map of the overall language. It is supplemented by thumbnails for each pattern, which are incorporated within a prose description tying all of the patterns together.

Provide Visual Illustrations Every pattern has a diagram illustrating itself with respect to an example from Critique. The diagrams have been designed to provide a quick snapshot of the pattern, so that it would be useful to someone casually browsing for the first time, or someone trying to quickly recall the purpose of the pattern. The captions are descriptive enough to gain an idea of the pattern just by looking at the figure and caption.

Eliminate Ambiguities Ambiguity among patterns is not a major issue here, since the patterns are quite distinct from each other by their nature.

Provide Instructions For Use The sequence in the next section provides a small set of instructions for using the patterns as a whole.

Observe Designers Interacting with Patterns Observing designers would be useful, but is beyond the scope of the present thesis.

11.3.2 Pattern Language Overview

Figure 11.7 shows the overall Planet language. The detailed Planet patterns expand on the organisational and high-level design patterns, which were described in Chapter 5. Once you have applied these high-level patterns to arrive at a specification for your `Multicultural System`, use Planet's detailed design patterns to design your software.

You begin by creating a `Global Data Model`:

▶ **`Global Data Model`** Encapsulate data required to support all cultures in a global data model, independent of the user-interface.

The `Global Data Model` is culture-independent, and it is now time to consider cultural variation. Encapsulate the user's preferences in a dictionary entity:

▶ **`Preference Dictionary`** Encapsulate all of the user's current preferences, whether culture-specific or not, in a single dictionary (i.e. key-value pairs) entity.

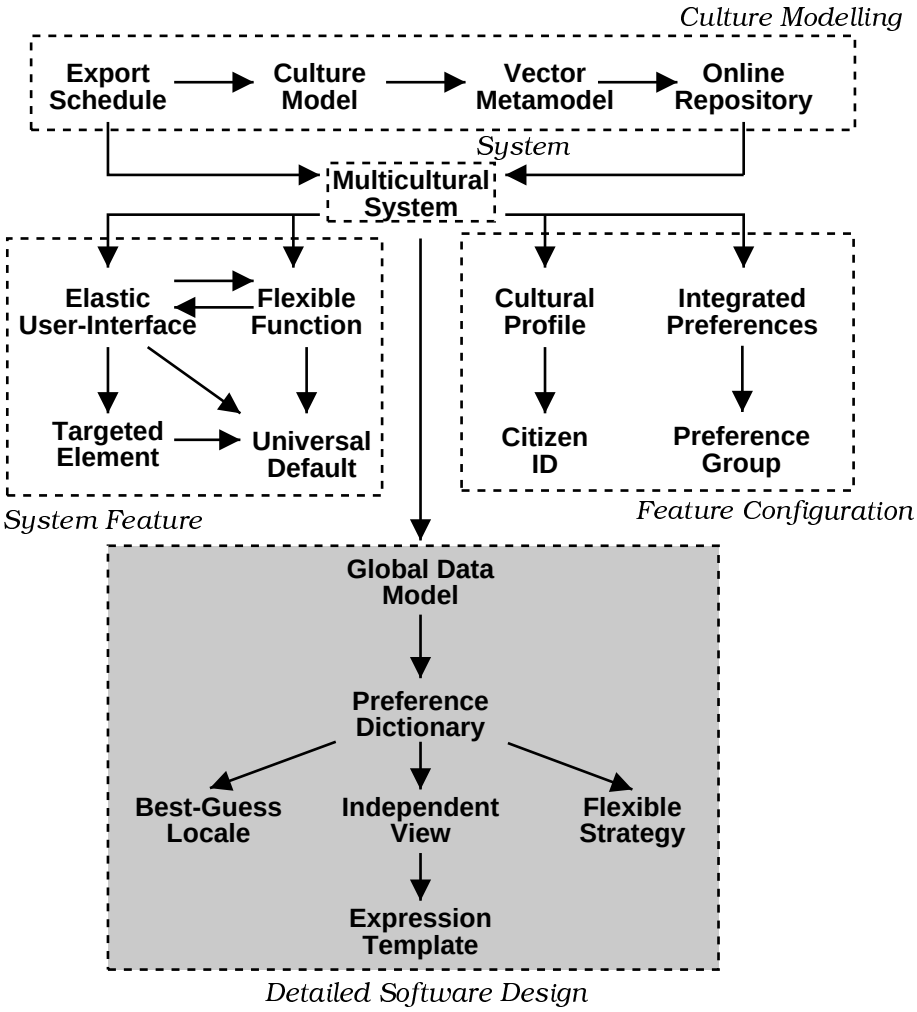


Figure 11.7. Map of overall Planet Language, with detailed software design patterns highlighted.

The `Preference Dictionary` will specify which resources can vary. You will need to provide the values of these resources for your supported cultures:

▶ **`Best-Guess Locale`** Create a "global culture" `Preference Dictionary` object which contains `Universal Defaults`. For each supported culture mentioned in your `Export Schedule`, create a `Preference Dictionary` object which overrides the global `Preference Dictionary`.

Two particularly important resources which can vary with culture are `Independent Views` and `Flexible Strategy`:

▶ **`Independent View`** Create one or more views of the data model, and store the user's preferred view within the `Preference Dictionary`.

▶ **`Flexible Strategy`** Create an abstract class which declares the interface for the algorithm, then create culture-specific implementations of this class.

The `Preference Dictionary` will specify which resources can vary. `Independent Views` often include messages whose format is culture-specific. So apply `Expression Template` in these situations:

▶ **`Expression Template`** Encapsulate each culture-specific expression in a template string. Store the template string in the culture's `Preference Dictionary`.

11.3.3 Descriptions of Detailed-Design Patterns in Planet

This section lists all of the detailed-design patterns in Planet.

Pattern 11-1. Global Data Model

Context: You have followed the high-level Planet patterns to arrive at a specification of functionality and user-interface for a `Multicultural System`.

Problem: How do you structure your program?

Forces:
- You need to allow for many parameters to be altered after you release the program. Since you are allowing users to adopt their personal preferences (see `Cultural Profile`), the number of possibilities is virtually infinite.
- The user should be able to change preferences while the program is running. Their data should remain the same, even though it will be processed or rendered in different ways.
- When you create a new feature, you should be able to tailor to specific cultures in your own time, in accordance with your `Export Schedule`.
- Some parameters will affect the user-interface. It is undesirable for development of new user-interfaces to hinder development of core functionality.

Solution: Encapsulate data required to support all cultures in a global data model, independent of the user-interface. The user-interface can observe the model and change itself or update the model when appropriate. The model, however, can be developed independent of the user-interface views. When you decide to store some extra data, you can gradually incorporate it into each of your culture-specific views. The model should encapsulate all data and operations on that data.

The format of the data should enable easy translation to localised formats, if necessary. It is usually okay to store data in the format of a well-supported culture, or the developer's culture. A word-processor can store page width in inches and convert to centimetres when necessary (although some caution must taken to prevent problematic rounding errors).

If conversion is difficult, the information must be stored in an abstract format. With text messages, for example, it is messy for the entire message to be stored in English. A translator must then perform string comparisons to determine the appropriate local variant. If the English ever changes, all of the localised code must be updated. Instead, such information can be stored as message IDs.

Examples: Critique uses the ArtModel class (Figure 11.8. It stores imageFile and ratings, two core objects which are necessary for all cultures.

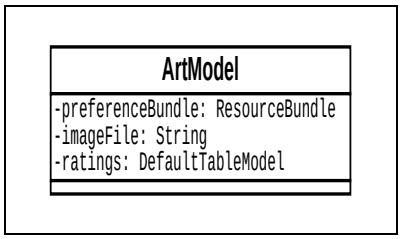


Figure 11.8. Global Data Model example: The ArtModel in Critique contains core data which is relevant for all cultures.

In the Vim text editor [225], implemented in C, users can edit in more than one window. The `buffer` data structure stores data relating to the text being entered, independent of user interactions. Data stored includes: the text itself, pathname of corresponding file, "dirty" flag (i.e. whether the current text differs from the corresponding file).

Universal Natural Language [220] is an ambitious attempt by the United Nations University to provide "a digital metalanguage" for representing information in a language-independent way. Once it is completed, it would form an ideal format for storing messages in a locale-independent manner.

Large-scale e-commerce websites use a database to track their inventory and keep information about manufacturers, recommendations, and so on. This database is a `Global Data Model` which can be queried by localised websites.

Remarks: This pattern, combined with `Independent View`, runs along the lines of the Model-View-Controller architectural style [148], which has been discussed previously in patterns literature [34, 43, 88]). In Planet, Model-View-Controller is not specifically advised; this would be unnecessarily constraining. The goal is to separate the presentation from the core data; whether MVC, PAC, or other architectural styles are used is a choice for the development team.

Resulting Context: Your data is stored in a well-contained module and manipulated with culture-neutral methods. You now need to create a `Preference Dictionary` to determine how it will be viewed and manipulated.

Pattern 11-2. Preference Dictionary

Context: You have created the `Global Data Model`.

Problem: There are many preferences which can change, some culture-specific and others culture-neutral.

How do you keep track of the parameters which a user can change?

Forces:
- `Culture Models` will change as developers learn more and the actual cultures themselves undergo change. A preference may be culture-neutral one day and specific to culture the next day, or vice-versa. You should not be hindered by such transitions.
- The user can tailor preferences to their own preferences, so culture alone is an inadequate specification of the current "preferences".
- You will be passing this information around between many modules. It should be as compact as possible.

Solution: Encapsulate all of the user's current preferences, whether culture-specific or not, in a single dictionary (i.e. key-value pairs) entity. Each parameter, whether or not culture-specific, has a defined key. The preference dictionary can be passed around and inspected whenever some code needs to perform a task which depends on the preferences.

The nature of preference values will vary widely. They may be a string representing some natural-language text, an image for a logo, or even a reference to a database table. Therefore, you will need to adopt a suitably flexible mechanism for your programming language.

Since some preferences may form a `Preference Group`, this may be a recursive class. For example, you may have a message preference dictionary inside a global preference dictionary. The message preference dictionary will contain numerous text strings (e.g. English words), but because the entire dictionary is stored as a single reference ("English Dictionary"), all of the strings inside it can be switched to a new language (e.g. to French words) simply by switching the reference to a new dictionary ("French Dictionary").

Examples: Java's ResourceBundle class is used by Critique for all preferences. The PreferenceDialog sets the preferenceBundle object, and it is then sent to the ArtModel, which propagates it to the view. The preferenceBundle is a dictionary with five keys (Figure 11.9).

In Java, the default values are also specified in the ResourceBundle. Thus, the main code for preferenceBundle is as follows (the structure from the ResourceBundle is adapted from the Java internationalisation tutorial [35]:

```
public Object[][] contents = {
        { "BackgroundColor", new Color(200,200,255)},
```

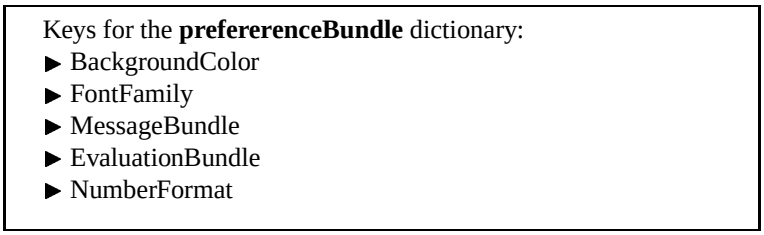


Figure 11.9. Preference Dictionary example: Critique's preferenceBundle has key-value pairs for all user-changeable parameters, whether culture-specific or not.

```
        { "FontFamily", new String("Courier")},
        { "MessageBundle", new MessageBundle_en()},
        { "EvaluationBundle", new EvaluationBundleEveryone()},
        { "NumberFormat", NumberFormat.getInstance(new Locale("en","US"))}
};

public Object[][] getContents() {
    return contents;
}
```

Note there is an EvaluationBundle and MessageBundle. These are nested ResourceBundles. The ResourceBundles are used in the following way [4]:

```
ResourceBundle preferenceBundle = ResourceBundle.
                  getBundle("critique.PreferenceBundle",currentLocale);
Font frenchFont = preferenceBundle.getObject("FontFamily"),Font.BOLD,30));
ResourceBundle messageBundle = (ResourceBundle)
                   preferenceBundle.getObject("MessageBundle");
String scoreMessage = (String) messageBundle.getObject("Score");
```

The Vim text editor [225] has dozens of options, including culture-related options such as right-left editing and alternative keymaps. From the user's perspective, these are defined in the same context as all other options, and therefore follow the `Integrated Preferences` pattern. At the code level, the options are mixed with other options in a global options file:

```
#ifdef RIGHTLEFT
EXTERN int      p_ari;          /* 'allowrevins' */
```

[4] This code is taken from various segments of the program

```
EXTERN int      p_ri;           /* 'revins' */
#endif
#ifdef CMDLINE_INFO
EXTERN int      p_ru;           /* 'ruler' */
#endif
```

Resulting Context: You have declared the parameters by which your software will vary. Now you need to define culture-specific information relating to the preferences with `Best-Guess Locales`. If your preferences vary according functionality performed, create `Flexible Strategies`. Complement your `Global Data Model` with `Independent Views`.

Pattern 11-3. Best-Guess Locale

Context: You have a `Preference Dictionary` which identifies parameters by which your software will vary.

Problem: Although the user can adjust preferences to their own settings, it is important to speed up the process with predefined `Cultural Profiles`.

How do you support `Cultural Profiles`?

Forces:

- When the user sets `Cultural Profile`, the software must somehow populate every parameter of the current `Preference Dictionary`.
- If you try to specify all adjustable parameters for all cultures, you can end up with a huge number of parameters which must be individually entered.
- If you try to specify all adjustable parameters for all cultures, localisation staff will often be duplicating work in their creation and maintenance efforts. If a group of cultures have many parameters in common, this information should not be lost.

Solution: Create a "global culture" `Preference Dictionary` object which contains `Universal Defaults`. For each supported culture mentioned in your `Export Schedule`, create a `Preference Dictionary` object which overrides the global `Preference Dictionary`.

The locale-specific vector is used as a `Cultural Profile` to simplify the user's task of setting preferences. According to `Cultural Profile`, the user should be given the option of choosing a culture. This is implemented by setting the "current" `Preference Dictionary` to the particular `Preference Dictionary` belonging to the user's desired culture. Parameters unspecified in the culture-specific vector revert to the global `Preference Dictionary` object.

The software will then be as specific as it needs to be. If some cultures do not care about background colour, simply leave it unspecified in their `Preference Dictionary`. The default back-

ground colour will be chosen, and a single change to that parameter will then propagate all cultures for which background colour needs no tailoring. For some parameters, it is fine if no culture-specific preferences are provided; these are culture-neutral preferences.

As mentioned in `Online Repository` cultures in the repository can be related to each other in a recursive manner. If desired, you can extend this concept to a hierarchical view of cultures, in which you progressively become more detailed. For instance, a French-Canadian user could use whatever French-Canadian preferences in the first instance, followed progressively by Canadian, then North American, then global. An alternative way to view this situation is to consider the algorithm for determining French-Canadian preferences: Begin by setting current preferences to global preferenceBundle. This preferenceBundle should define settings for each variable. Then, for all of the variables which are specified in the North American preferenceBundle, set the current preferences to these variables. Do the same for the Canadian preferenceBundle, then the French-Canadian preferenceBundle. The current preferences will therefore end up as specific to French-Candadian culture as possible.

Examples: Critique supports French and English locales. The English preferenceBundle overrides the default MessageBundle and EvaluationBundle, while the French preferenceBundle overrides MessageBundle, EvaluationBundle, and also BackgroundColor for demonstration purposes:

```
public Object[][] contents = {
        { "BackgroundColor", new Color(100,200,200)},
        { "MessageBundle", new MessageBundle_fr()},
        { "EvaluationBundle", new EvaluationBundleTeacher()},
};
```

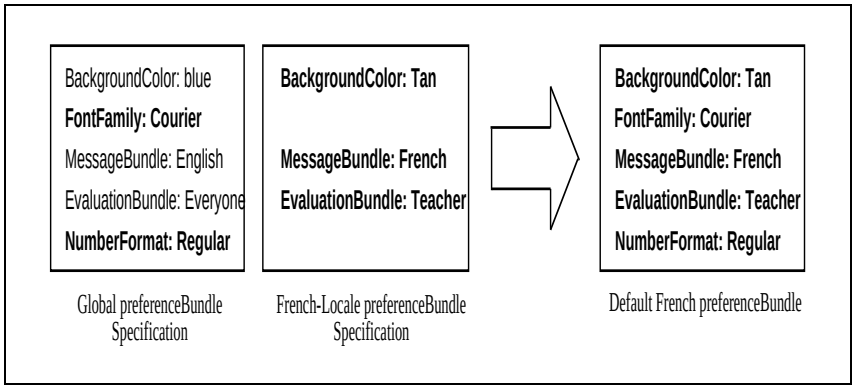


Figure 11.10. Best-Guess Locale example: The French preferenceBundles overrides the global preferenceBundle. The resulting preferenceBundle is the default preferenceBundle for French users.

As explained in the examples for Preference Dictionary, the Vim editor [225] has culture-specific options integrated with other options. Vim also exemplifies Best-Guess Locale because users can start Vim in Hebrew mode or Farsi mode, which has the effect of defining certain options with the appropriate initial values. Below is the code which checks for the Hebrew command-line flag, '-H'. It sets the current window's rightleft (w_p_rl) flag and Hebrew Keyboard map (p_hkmap) flag to true.

```
case 'H': /* "-H" start in Hebrew mode: rl + hkmap set */
#ifdef RIGHTLEFT
    curwin->w_p_rl = p_hkmap = TRUE;
```

Resulting Context: The user can quickly set up preferences according to their own culture.

Pattern 11-4. Independent View

Context: Your Preference Dictionary suggests different views will be provided to the user.

Problem: The Global Data Model has no user-interface component.

How does the user view and modify the Global Data Model?

Forces:
- Your Global Data Model contains a lot of data. Cultures will want to see different layouts and orderings, as discussed in Elastic User-Interface.
- The default views for some cultures should hide data which is irrelevant or offensive.
- Sometimes, the user wants to see the same data according to two different views. This could occur if they are shifting their own "cultural mode" as they deal with different type of information (see Citizen ID), or if they are trying to determine how someone else sees the data.

Solution: Create one or more views of the data model, and store the user's preferred view within the Preference Dictionary. You will need a standard view interface, so that different views can be plugged into the user-interface in a standard way.

The different view classes can have different layouts and may hide some information if it is irrelevant or offensive to some cultures. Of course, this flexibly provides only some of the variation in the user-interface. A concrete view class is still parameterised according to other preferences. It needs to query the Preference Dictionary for Targeted Elements like messages, logos, and background colours.

Remarks: With users from multiple cultures, it is desirable to can show two views of the same data at the same time. This is possible with an MVC approach, but may be more difficult in PAC and other styles.

Examples: Critique uses the MMVC framework (Chapter 10) to separate the `Global Data Model` from its views. The ArtModel in Critique is a `Global Data Model`. ArtViewEveryone and ArtViewTeacher constitute independent views (Figure 11.11). The model broadcasts its changes to any registered listener, and the views register themselves as listeners.

ArtViewEveryone is the global default and also the English default, and shows details by all raters. ArtViewTeacher shows only the teacher's view, and is the default French view. This is implementing by making the French preferenceBundle(preferenceBundle_fr) specify EvaluationBundleTeacher as the value for EvaluationBundle. EvaluationBundleTeacher, in turn, specifies ArtViewTeacher as the value for ArtView.

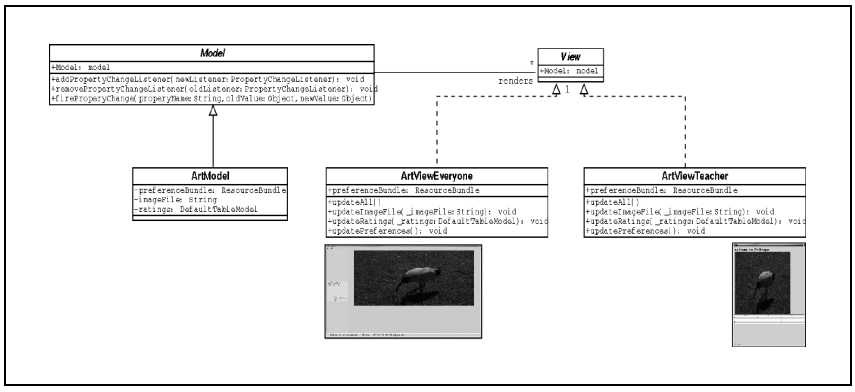


Figure 11.11. Independent View example: Critique's ArtViewEveryone and ArtViewTeacher are both capable of rendering the ArtModel and the ArtModel does not depend on these classes.

As explained in `Global Data Model`, Vim [225] stores data separately from the user-interface. The main structure which does store the user-interface information is `window`, which most importantly stores a `buffer` of text (the data model). It also contains attributes which track its own height, the portion of the buffer being displayed, and the cursor position.

Resulting Context: The user receives a tailored presentation and the `Global Data Model` is not coupled to any presentation code.

Pattern 11-5. Expression Template

Context: An `Independent View` needs to output expressions which have culture-specific formats.

Problem: How can you incorporate culture-specific formats into the `Preference Dictionary`?

Forces: • Your software probably needs to output text, and in some cases, parse user-inputted text.

- Grammar varies widely, e.g. some cultures say “Subject Verb Object” and other cultures say “Subject Object Verb”. You cannot just write global code which does something like:

```
print "$subject has sent you a message";
```

- Grammar is not the only reason why expressions vary by culture. There are many formatting concerns (Table 5.1): numbers, date/time, addresses, telephone numbers.
- You could treat the different expression formats as a situation where different code is required, and adopt `Flexible Strategy` for each expression. But this would lead to an explosion of classes in most applications, especially if you did this for every affected text message.

Solution: Encapsulate each culture-specific expression in a template string. Store the template string in the culture’s `Preference Dictionary`. Expression templates enjoy native support in many languages, from Fortran’s `FORMAT` to C’s `printf/scanf` to Java’s `Format`. In most cases, it is possible to store the template string in a variable for later use.

Examples: The number format is flexible in Critique (Figure 11.12), varying in number of decimal places. The NumberFormat is stored as an entry in the `Preference Dictionary`. This is not particularly relevant to culture, though the same mechanism could allow for culture-specific formats like “302.100,2” instead of the more common “302,100.2”. Formats for addresses and telephone could be handled in the same way.

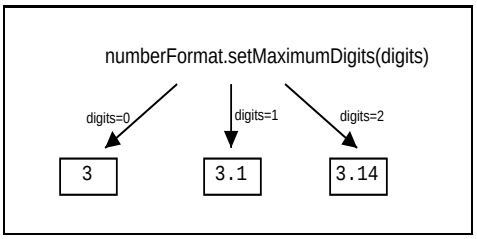


Figure 11.12. Expression Template example: Critique’s NumberFormat is stored in the preferenceBundle. It determines how many decimal places are shown for the overall score, and the same mechanism could also be used to show culture-specific number formats as well as formats for phone numbers or addresses.

Mozilla is a web browser supporting international audiences [5], implemented in C++. It contains a class which provides a common interface for date and time formats, nsIDateTimeFormat. A key method is `FormatTime` (which is virtual because it has OS-specific implementations):

[5] Mozilla was originally the open-sourced Netscape

```
NS_IMETHOD FormatTime(nsILocale* locale,
    const nsDateFormatSelector dateFormatSelector,
    const nsTimeFormatSelector timeFormatSelector,
    const time_t  timetTime,
    nsString& stringOut) = 0;
```

The implementations of this method use libraries which format time according to locale.

Resulting Context: You have a mechanism for encapsulating a tailored format into a single object, which can be placed in the `Preference Dictionary`.

Pattern 11-6. Flexible Strategy

Context: Your `Preference Dictionary` suggests the need for flexible functionality.

Problem: How can you incorporate culture-specific functionality into the `Preference Dictionary`?

Forces:
- Your software needs `Flexible Functions` to deal with cultural variation (e.g. taxation rules, operator permissions).
- This type of variation often requires tailored code. Your `Preference Dictionary` can easily specify a string or an image, but how does it specify a piece of code?

Solution: Create an abstract class which declares the interface for the algorithm, then create culture-specific implementations of this class.

Remarks: This pattern is essentially Gamma et al.'s *Strategy* [88] pattern, re-stated in the context of Planet's scope and related patterns.

In non-OO languages, it might be possible to store a pointer or reference to a culture-dependent function.

Examples: In Critique, there is a `Flexible Function`: the score can be based on the teacher's ratings or on everyone's ratings. This is implemented as a `Flexible Strategy`. Scorer defines the interface for accessing the `calculateScore` method. The `evaluationBundle` stores a reference to the class that implements the desired scoring algorithm.

Gamma et al. list several examples of `Strategy`, although these are for general usage rather than culture-specific.

Resulting Context: You have a mechanism for encapsulating tailored code into a single object, which can be placed in the `Preference Dictionary`.

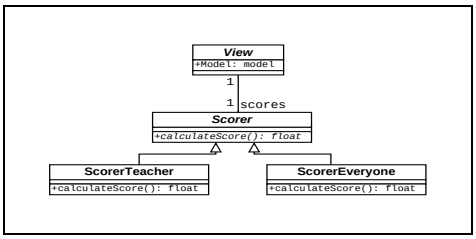


Figure 11.13. Flexible Strategy Example: Critique's ScorerEveryone and ScorerTeacher allow the scoring strategy to be implemented a flexible manner, determined by the user's needs.

11.4 Discussion

The extended Planet language contains patterns which implement the type of `Multicultural System` which was envisioned in creating the high-level Planet patterns. The high-level patterns suggest certain capabilities which are resolved jointly by the network of detailed patterns. That is, the high-level patterns do not map one-to-one with detailed patterns. Instead, the detailed patterns work with each other to produce the kinds of features which are required. There are *some* close mappings, i.e. `Culture Model` $\rightarrow$ `Best-Guess Locale`, `Elastic User-Interface` $\rightarrow$ `Independent View`, `Flexible Function` $\rightarrow$ `Flexible Strategy`. Even with these patterns, though, the patterns emphasise their relationship to other detailed design patterns. A relatively generative process is able to be offered.

Interaction among abstraction levels is perhaps the most powerful application of patterns in HCI. It provides a common ground for stakeholders with different roles and qualifications, an important achievement for any HCI-related techinque. Most Planet patterns are simple enough for users and other non-technical staff to contribute to. The organisational patterns focus on the development of the Repository, which contains information pertinent to the choices of features and user-interfaces. The patterns for these entities detail how the organisational patterns are used to arrive at tailored solutions. Finally, the patterns here show how the software is actually implemented.

The work presented in this chapter consolidates several concepts which have been introduced throughout this thesis:

Patterns, Languages, and Constraints (Chapters 3,4) It has been argued that constraining the scope of patterns in some aspects can allow for a coherent relationship among patterns, and yields some expansion of scope in other directions. Planet demonstrates that a set of patterns which focus specifically on flexible systems can unify organisational process, high-level design, and detailed software design.

High-level Planet pattern language (Chapters 5,6) The detailed design patterns presented in this chapter are a natural extension of the high-level patterns and supporting repository.

Observational study of safety-usability studies (Chapter 7) By observing people using the safety-usability patterns, it was possible to speculate on some guidelines for pattern language construction which may help designers. The detailed Planet patterns, which were created after the safety-usability study, follow these guidelines.

Support for generic tasks with the MMVC Framework (Chapters 8,9, 10) The work in Chapters 8 to 10 served to demonstrate that detailed design patterns can facilitate usability, and bridge gaps between high-level constructs (i.e. generic tasks) and detailed software design. The detailed Planet patterns are the next logical step: rather than moving from generic tasks to software design patterns, they allow for a process incorporating patterns at each stage. A secondary benefit of the MMVC framework is that it was applied here to form the separation between `Global Data Model` and `Independent Views`.

In this chapter, the patterns are not constrained by programming language or architecture style. This was useful because it allowed for a greater number of examples to be drawn upon: the main examples are Critique (Java), Mozilla (C++), and Vim (C). All three differ in their purpose, their programming language, and their architecture. But because they share the common feature of internationalisation-preferences, each of them has features which make useful examples.

While it has been useful to take examples implemented in different programming languages and architectures, it would also be interesting to consider construct a language from example applications which are very similar to each other in these areas. This would help to further explore the theme that constraints can be advantageous. It would require a reasonable amount of work, since a few applications would need to be built within that framework. However, an organisation with a similar suite of internationalised applications could probably perform this kind of work, and derive benefits from it once achieved.

Part IV

Conclusions

Chapter 12

Conclusions

12.1 Introduction

For HCI techniques to be applied effectively, they must be incorporated into the overall SE process. Several approaches to HCI-SE integration were surveyed in Chapter 2, but none of these incorporated a structured approach to reusing successful design concepts. Design reuse is an essential step towards the establishment of HCI as a mature discipline. In one form or another, design reuse is being applied by more mature disciplines such as civil engineering, organisational management, and medicine. Given that user responses are usually unable to be predicted from first principles, HCI stands to benefit significantly from techniques which do not require designers to reinvent the wheel whenever a design issue emerges.

In some respects, HCI knowledge reuse already exists in the form of guidelines. However, there are numerous problems with this approach; high-level principles suffer from vagueness and conflict with each other. Low-level rules can be too specific, also conflict with each other, and are not practical for high-level design. This thesis has studied design patterns and other alternatives to design guidelines. As stated in Chapter 1, the aims of this thesis have been:

- To discover and evaluate techniques which facilitate reuse of existing *high-level* design features that have contributed to usability.
- To discover and evaluate techniques which facilitate reuse of existing *detailed* design features that have contributed to usability.
- To investigate how these techniques may be combined to improve their overall effectiveness. In particular, to consider whether it is possible to produce a knowledge base containing both high-level design knowledge and detailed design knowledge, such that the detailed design knowledge guides developers on the implementation of features suggested by the high-level design knowledge.

These conclusions outline the contributions of this thesis and explain the extent to which they have

achieved these aims. Opportunities for further research are also explored.

12.2 Contributions in this Thesis

This thesis has exposed several approaches to design reuse which promise to improve the maturity of the HCI discipline and incorporate it into the SE process. Examples of design patterns, software frameworks, generic tasks, and repositories of user knowledge, have been created and studied, and all demonstrate the approaches could be recruited for effective application by software developers.

Primarily through these products, the thesis has contributed to HCI and SE theory in the research areas listed below:

1. Classification of HCI Patterns
2. Relevance of “language” in “pattern language”
3. Interdisciplinary nature of pattern languages
4. Presentation and usage of pattern languages
5. Capture and usage of generic tasks
6. Patterns and frameworks in HCI
7. Auxiliary contributions arising from Planet, Safety-Usability Patterns, and MMVC

These contributions are elaborated in the following sections. Where relevant, these sections also indicate relevance to the thesis aims.

It is instructive to consider the concrete products which helped to develop and demonstrate these contributions. The major products arising from the thesis are as follows:

1. Planet Pattern Language (Chapter 5, 11).
2. Critique Application demonstrating Planet (Chapter 11).
3. Safety-Usability Patterns and results from design study (Chapter 7).
4. Generic task list and results from brainstorming study (Chapter 8).
5. MMVC Framework — reusable software components, source-code generation tool, design patterns, and example applications (MARCO, Prescribe) (Chapters 9, 10).
6. Online Repository of Cultural Factors (Chapter 6).

12.2.1 Classification of HCI Patterns

At the commencement of the research embodied in this thesis (March, 1997), there was almost no material available on patterns for HCI. A CHI '97 workshop discussed this concept [14] and Casaday proposed several pattern styles at the same conference [39], but there were no wide-scale collections in existence. After deciding that HCI patterns would be a viable area for research, Chapter 4 expanded on Casaday's work to explore in detail what types of patterns might be possible for HCI. Thus, work on patterns in the thesis began by establishing a classification of patterns for usability [145], and led to the dimensions discussed in Chapter 4. The first aim of this thesis concerns discovering and evaluating techniques for high-level design reuse; this classification and evaluation was a "breadth-first" achievement of these aims.

The analysis gave examples for individual patterns, rather than attempting to provide sketches for entire languages. It was possible to speculate on how a language might appear by looking at an individual pattern, although this is probably not as effective as if a collection had been sketched. Nevertheless, there are now a wide range of pattern collections which have been produced in recent years, and the classification was used to summarise a number of prominent projects, in Chapter 4.

12.2.2 Relevance of "Language" in "Pattern Language"

A key theme has been the emphasis on language. The three pattern collections in this thesis — Planet, Safety-Usability Patterns, MMVC Patterns (part of the MMVC framework) — have each been prepared with an emphasis on high coherence within the language. This means the patterns share a common set of underlying principles, effectively delegate to each other, and support incremental ("piecemeal") development.

A strong sense of *language* has been achieved by sacrificing generality. Planet deals specifically with users from different cultures. Its patterns are inter-related, and cover concepts ranging from organisational process to detailed software design. MMVC supports certain generic tasks under a specific user-interface architectural pattern (MVC), and relates to a set of reusable software components. Safety-Usability Patterns are concerned only with design of safety-critical systems. It was noted that the language is not as well-integrated as Planet, representing something like a set of related "mini-languages". There is still a large degree of inter-relationship among many patterns, and this is a consequence of the specific nature of the language, i.e. certain features recur in safety-critical systems and often relate to each other in the same ways.

Each of the three example languages corresponds to a different thesis aim — Safety-Usability patterns are high-level patterns, MMVC patterns are detailed design patterns, and Planet is a combination of both. The thesis demonstrates how pattern languages can support reuse at each level. The demonstration of the relationship between constraints and pattern languages is also a general contribution to pattern theory.

12.2.3 Interdisciplinary Nature of Pattern Languages

MMVC demonstrates that detailed software design patterns can address usability. MMVC combines reusable components and a pattern language to guide their application in a user-centered manner. This is a model of patterns which allows HCI knowledge to be stated in terms programmers can deal with. It does not replace the role of the HCI specialist, because usability consideration must begin earlier than detailed software design. However, the approach does help to ensure usability is considered at each stage. By following this approach, patterns can help software developers balance usability with other attributes such as reliability and performance, hence it is a means of bridging the SE-HCI gap which has been a key motivation of this thesis. It also facilitates consistent user-interface presentation and user-computer dialogue. This kind of consistency could not be provided by simply stating the principle “Design consistently”. It might be offered via detailed design rules, but such rules can be difficult for programmers to follow and interpret, and so cannot adequately address the technical challenges that programmers must also face.

Thus, Chapter 9 uses MMVC patterns to show how patterns can improve usability at a detailed level. Likewise, Safety-Usability patterns address high-level usability issues — task management, task execution, information presentation, and machine control. With Planet, the thesis shows how a single pattern language can cross abstraction boundaries. The language begins with organisational patterns, leads into high-level specification patterns, and then shows how these can be implemented with detailed software design patterns. This is an interdisciplinary approach to patterns which has the potential to render pattern languages into artifacts which can be shared by the overall development organisation, similar to Harris and Henderson’s futuristic vision showing how patterns may eventually play a key role inside organisations [102].

These ideas help to achieve the second aim of the thesis, relating to detailed software design. The background to this particular aim was the notion that detailed software design can impact on usability. MMVC shows how advice to detailed software designers can indeed have an impact on usability. The interdisciplinary nature of Planet is a demonstration of the third aim, relating to the combination of these approaches. The strength of this argument could be extended by studying interdisciplinary teams using these languages, in a similar manner to the way software engineering students were studied using Safety-Usability patterns in Chapter 7. It is likely that useful insights could be gained by observing practitioners from different backgrounds sharing a pattern language to solve a design problem.

12.2.4 Presentation and Usage of Pattern Languages

Several interesting observations emerged from the controlled experiment involving Safety-Usability Patterns (Chapter 7). It was possible to propose a set of guidelines which authors of pattern languages can follow to improve usability of their languages. These guidelines were later used to create the detailed Planet patterns, and could be generalised to other software and HCI pattern languages. As such, they relate

at some level to all aims of this thesis. A further study based on patterns conforming to the guidelines in mind would help to validate the findings.

Several implications for developers using pattern languages were also noted in the study, and could also benefit from confirmation in further empirical studies.

12.2.5 Capture and Usage of Generic Tasks

The first aim of the thesis concerned reuse of high-level design features, and patterns are not the only way to achieve this; generic tasks are another approach (Chapter 8). The thesis has shown how a set of generic tasks can be systematically captured and used for development purposes. The brainstorming study suggested that the approach is easy to learn and a majority of subjects were comfortable with the approach. The subjects were generally able to make useful contributions as a result of being introduced to the generic tasks. However, the guidelines came after the study was performed; further studies would be required to validate their efficacy, perhaps by assessing the quality of those tasks which arise from the generic tasks.

The generic tasks were also used as an input to the MMVC framework. This approach suggests a way to combine the high-level technique of task analysis with detailed software design. However, the generic tasks are not design patterns themselves, so the approach led to the question of whether it would be possible to use patterns at each stage. The complete version of Planet suggests that this is indeed feasible.

In their brainstorming capacity, generic tasks are a contribution to requirements elicitation within HCI. The application of generic tasks to software frameworks is a further contribution to SE-HCI integration.

12.2.6 Patterns and Frameworks in HCI

The MMVC framework takes into account Johnston's view of a framework as being a collection of reusable components and patterns [123]. Reusable components are typically created with certain usage mechanisms in mind; the patterns act as a way of documenting them. MMVC shows what this means in an HCI context. It also adds a source-code generation tool to compensate for certain necessary repetition of code. The Prescribe application exemplifies the framework and was created alongside it. Part of the framework was subsequently used to create the Critique application, and was helpful in producing the program relatively quickly and in a consistent manner.

The MMVC framework shows how patterns can be supplemented with other tools to encourage usability at the detailed software design level, which relates to the second thesis aim. Software designers face enough challenges in the absence of usability issues; any support for usability at this level will facilitate its consideration, and reusable components and code generators constitute additional support.

12.2.7 Auxiliary contributions arising from Planet, Safety-Usability Patterns, and MMVC

The Planet and Safety-Usability pattern languages, as well as the MMVC framework, led to several contributions which, while not directly connected to the thesis aims, are still noteworthy. Four significant auxiliary contributions are outlined here.

Reuse of Knowledge about Users

A repository of cultural information was suggested by Planet and subsequently implemented as a prototype project. It suggests a type of knowledge reuse which differs, yet also complements, design patterns. It differs because it captures a wide range of knowledge and also because it has been studied with a heavy focus on the tool; the corresponding pattern is termed "Online Repository" and the concept cannot be separated from technology. Patterns, on the other hand, are largely technology-independent. Even if they are accessed electronically, they are fundamentally a static description consisting of text and images. The Planet patterns expose how the repository can be used, at least for the case when the online repository captures cultural information (as is the case for the prototype discussed in Section 6).

Requirements for Internationalised Software

An important side product of Planet is the analysis of cultural factors. Developing Planet and the online repository involved a comprehensive search of internationalisation literature and internationalised software [147]. This led to the listing of many relevant cultural factors and also a number of issues for practitioners and researchers to consider.

HCI Guideline Summary and Safety-Critical Implications

As background work for the Safety-Usability Patterns, it was important to explicitly state the language's underlying principles. These principles were derived by initially summarising principles contained in a set of well-known style guides, and then analysing the safety-critical principles. The result is a concise statement of the implications that safety-critical systems hold for general HCI principles.

Refinement of Model-View-Controller (MVC) Architecture

A further side product of the project was the refinement of the MVC architecture. Aside from its relevance to generic tasks and the MMVC framework, the work performed while preparing MARCO makes some useful contributions to software design theory about limitations of MVC and how they can be addressed [148].

12.3 Future Directions

This thesis has explored techniques for design reuse in HCI and SE. The work complements conventional forms of design reuse, namely principles and design rules, with concepts such as design patterns, online knowledge repositories, and software frameworks, all with a focus on addressing the combined needs of usability and other software attributes. A large amount of work on patterns in HCI has evolved in parallel with this project, as was discussed in Chapter 4. Sufficient work has now been done to understand what types of patterns can improve user-centered design. This work, in turn, opens up many new questions. This section concludes the thesis by outlining the major remaining issues.

12.3.1 Supporting Adoption of Patterns in Industry

While HCI patterns can benefit researchers by improving expressivity, their primary purpose is to facilitate software development. This degree of maturity has not yet been achieved. Patterns for HCI have mostly been created by researchers and not deployed in industry. In contrast, software design patterns have been created by a healthy mix of academics and industry, and are known in both contexts. To progress the field of HCI, patterns will first need to be evaluated in an industrial context. The tools in this thesis would need to be revised in co-operation with industry for practical uptake, but more importantly, they act as models for the type of design reuse which is possible. This model could be used to generate patterns and related tools which are specific to a particular community's needs. The guidelines arising from the Safety-Usability Patterns experiment should also assist industry, even though more experimentation is required to enhance confidence. To gain widespread acceptance, it is particularly important to provide practical tools which are familiar to developers. MMVC provides a model for such an approach; a set of reusable components ought to give developers a desirable productivity benefit, and the patterns complement the components. As well as studying initial system development, it would be interesting to consider system evolution — how well do the patterns support piecemeal growth on real-life projects? A longitudinal study, looking at the creation and ongoing maintenance of a software system designed according to patterns, is desirable. Another issue that could be considered in an emprical study is the difference between loose collections and well-integrated languages. The work in this thesis would predict that the latter form would support smoother ongoing development, and it would be useful to investigate the issue in an industry-based environment.

A further challenge for industry adoption is the development process. For both software design patterns and HCI patterns, there are questions about how they integrate with other tools and processes. There are already useful approaches such as MUSE [135] and Boehm's spiral lifecycle [22, 23] which help to address these issues and this area of HCI-SE research will doubtless undergo further development. Planet provides some support, since it is interdisciplinary and covers process issues. However, a language like Planet is not, and should not be, detailed enough to specify a complete process architecture. This leads to the question

of managing the general process and the process implied by the pattern language. One step towards the solution is probably to tailor the language to a particular project, an activity which Alexander encouraged long ago [4] and is still pertinent. It might also be possible to relate patterns to particular methodologies, such as patterns of the various stages in the MUSE methodology. However, this is a complex issue. Relating patterns, with their own implied processes, to the overall development process is a prerequisite to successful pattern-led work, and an important topic for future research.

12.3.2 Combining Pattern Languages

Related to the issue of tool and process integration is the question of how pattern languages themselves may be combined with each other. In software patterns literature, it is relatively common for patterns in one language to reference those in another language. But what if a developer wants to use two languages for a single application? A global safety-critical application, for example, might benefit from both Safety-Usability Patterns and Planet. This thesis has largely avoided the issue of combining pattern languages because the issue has not been studied much in general patterns literature, and even the goal of producing a single, well-formed, pattern language entailed many research issues which have not been fully addressed in the literature. Perhaps a significant part of the answer is that the patterns must be applied in a commonsense manner and can be tailored for a particular project. However, it would be resource-intensive to continually tailor languages. A "pattern repository" of the kind being pursued by Van Welie [224] seems likely to help developers to draw on patterns from many authors. The risk is that the coherence offered by a pattern language will be lost, but this alone should not stop an effort which will help practitioners consolidate the large number of documented HCI patterns. The pattern languages in this thesis may act as examples which can facilitate the organising and documenting of such a repository. Exactly how this could occur is a topic for further research, but one example might be the documentation of "meta-languages" which provide logical structure to a number of individual patterns customised for some purpose. A meta-language would be a reusable template for a project-tailored language which leverages from all HCI patterns in the repository. This thesis may also help developers of a pattern repository understand what aspects need to be stored, e.g. a pattern language often has a map showing delegation; a collection or an isolated pattern does not.

12.3.3 Addressing the Tension between Generality and Specificity of Pattern Languages

An important theme throughout this thesis is the tension between the rich guidance afforded by highly-specific patterns, and the broad applicability offered by general patterns. This theme could be explored further by extending the pattern languages of this thesis in both directions, i.e. further constraining the languages constraints and, in alternative work, generalising the languages.

The Safety-Usability Patterns and Planet have been constrained at what might be labelled a medium level. They are somewhat specific because they apply to certain specialised requirements (safety and international users, respectively). They are somewhat general because they are largely independent of software architecture. The MMVC language is mostly constrained with respect to detailed design, i.e. the constraint of an MVC-based architecture, and one which utilises the MMVC reusable components.

To investigate the possibility of increased constraints, it would be instructive to revise Planet to incorporate detailed design patterns like those in MMVC. Planet already has the benefit of strong interdisciplinary support, including guidance to designers of detailed software. However, it does not focus on a specific framework such as the C++ Locale framework [131], and this means it can be shared and evolved by anyone working on internationalised systems. For an individual development organisation which does deal with specific frameworks, it would be useful to tailor Planet in a way which provides detailed instructions about how the framework is used and how it fits in with the overall process. This would mean adopting patterns like those inside MMVC, which address detailed software design, using a specific architectural style, while still acknowledging usability issues.

However, the utility of more universal patterns cannot be ignored. In software design, Gamma et al.'s patterns [88] have made a large impact despite being quite general in nature. A general language is not only more broadly applicable; it can be used and refined by a larger number of practitioners on the basis of collective experience. Therefore, it is desirable to revisit the more broadly-scoped HCI pattern languages in light of the present work performed here. This may involve feeding the results of this thesis into the development of a patterns repository, as described above. It may also involve generalising the pattern languages of this thesis. For instance, Planet could be transferred to focus on the general problem of supporting different user types or market segments, regardless of internationalisation issues. The Safety-Usability patterns might be transferred to cover high-integrity systems like banking software and inventory control. MMVC might consider the general issue of data-view separation, rather than the specific MVC approach. In each of these cases, the applicability would broaden, though the quality of advice might diminish somewhat. The purpose of developing and evaluating pattern languages like this would be to find the right balance between generality and specificity.

12.3.4 Creating Online Repositories of User Knowledge

The thesis also covered aspects of reuse other than patterns. In particular, the repository of knowledge about users is an idea which could benefit from further work. The prototype discussed in Chapter 6 needs to be developed, populated, and used. While this prototype related to cultural factors, a repository of that nature could actually generalise to consider other user characteristics, just as Planet could also consider user characteristics other than culture. Further work might consider how other repositories would be designed and used. Since the repository's usage is described via Planet, a project like this would have implications for other pattern languages which rely on storage and retrieval of related knowledge.

Concluding Remarks

Patterns are still relatively new, but the amount of research over the past few years suggests they hold much promise for the discipline. It will be appropriate for the research issues mentioned above to be considered while the approach is integrated into mainstream HCI work. The work in this thesis suggests that the HCI community can use patterns to express ideas about many HCI concepts. Researchers often provide guidelines to express advice about knowledge they have discovered; patterns would be a suitable way to complement such guidelines. By their nature, patterns force authors to consider how their guidelines apply in realistic design contexts and how trade-offs must be made among conflicting guidelines. At this time, HCI patterns have mostly been studied as an isolated research field, with researchers asking exactly what they are and where they can be used. But perhaps their greatest potential will be as a tool for communicating HCI and SE advice among researchers and practitioners alike. It is intended that the present work contributes to the HCI design aids of the future.

Appendices

Appendix A

Deriving Properties of Usability

A.1 Introduction

This appendix describes how a set of desirable usability properties were derived. It is beyond the scope of this thesis to describe the work performed here in detail; more information is available in Mahemoff and Johnston, 1998 [146].

A.2 Usability Properties

Guidelines for the following platforms, toolkits, or paradigms were chosen:

- Apple Desktop Interface[Apple-87][7],
- IBM Common User Access[IBM-92][115],
- MS Windows 95[MS-95][165],
- OSF/Motif[OSF-93][179],
- Sun Open Look[Sun-89][208].

Each guideline in each publication was recorded (with a few exceptions, such as guidelines related to development process). These were then grouped to form a summary of guidelines, as shown below. For style guides which use higher-level terms to group several properties together, the secondary terms are located in whichever category is most appropriate, and are followed by "(S)".

1. Consistency *Consistency[Apple-87], Modelessness[Apple-87], Perceived stability[Apple-87], Making the user-interface consistent[IBM-92], Maintaining continuity within and among products[IBM-92] (S), Sustaining the context of a user's task[IBM-92] (S), Using modes judiciously[IBM-92] (S), Consistency*

[MS-95], Keep interfaces consistent[OSF-93], Consistency[Sun-89], Mouse and keyboard usage[Sun-89] (S), Placement and naming of controls[Sun-89] (S), Standard conventions[Sun-89] (S).

Consistency implies that set conventions are followed within and between applications. This means that related objects exhibit similar appearance and behaviour and input devices are interpreted in a consistent manner. Temporal consistency—or stability—is also recommended by most style guides, and this is often achieved by modelessness. Several reasons are given for consistent software: improving the learning curve; making the system more predictable; ensuring that switching between applications is a smooth activity.

2. Familiarity *Metaphors from the real world[Apple-87], Plain language[Apple-87], Making objects concrete and recognisable[IBM-92] (S), Using visual metaphors[IBM-92] (S), Use real-world metaphors [OSF-93], Familiar framework[Sun-89] (S).*

Familiarity, as the guides describe it, may be viewed as consistency with the real world. Thus, they recommend familiar features in order to help users build on existing knowledge, make the interface more approachable, and improve the learning process. Principles which endorse metaphors and user-oriented language contribute to familiarity.

3. Simplicity *Displaying descriptive and helpful messages[IBM-92] (S), Obvious class distinctions[IBM-92] (S), Simplicity[MS-95], Clearly labeled controls[Sun-89] (S), Make navigation easy[Sun-89] (S), Simplicity[Sun-89].*

Designing for simplicity means making an application straightforward to the user, so that they are not overwhelmed by details. It also suggests that they have an adequate mental model of the software's workings, and are able to access operations easily. A key motivation for this principle is that novices should be able to perform typical tasks with a minimum of difficulty, and simplicity should also make an application easier to learn. Appropriate visual design, familiarity, consistency, and task support all contribute to simplicity. However, simplicity may be a hindrance for experienced users (see "Efficiency", below).

4. User control and feedback

a. Direct manipulation *Direct manipulation[Apple-87], See-and-point (instead of remember-and-type) [Apple-87], User control[Apple-87], Reducing a user's memory load [IBM-92], Placing a user in control of the user interface[IBM-92], Directness[MS-95], User in Control[MS-95], Give the User control[OSF-93], Allow direct manipulation[OSF-93] (S), Provide output as input[OSF-93] (S), Working with objects [Sun-89]. (S)*

Since the selected style guides are all intended for WIMP (Window, Icon, Menu, Pointer) operating systems, it is not surprising that direct manipulation principles enjoy frequent discussion. In many ways, the major goal of this paradigm is to place the user in control, instead of allowing the computer to dictate what the user must do at each point in time. Furthermore, direct manipulation provides familiarity in that people are used to working with tools and objects. Because the objects and tools are represented on the interface, direct manipulation also emphasises recognition rather than recall.

b. Flexibility *Allowing a user to customise the user interface[IBM-92] (S), Keep interfaces flexible[OSF-*

93] (S).

Demand for flexibility is driven by the observation that there is a diverse range of users and usage environments. Thus, most style guides suggest that there should usually be more than one way to perform a task. [IBM-92] goes so far as to require that every task be possible using either the keyboard or the mouse. If users are to be in control of the interaction, it also follows that they should be able to customise the interface themselves, a suggestion also offered by some of the guides.

c. Feedback *Feedback and dialog[Apple-87], WYSIWYG (what you see is what you get) [Apple-87], Providing immediate actions, feedback, and reversible actions[IBM-92] (S), Feedback[MS-95], Communicate application actions to user [OSF-93], Give the user feedback[OSF-93] (S), Provide rapid response [OSF-93] (S).*

The style guides recommend that users' actions result in rapid feedback, in a form which is consistent and understandable to users. This way, users can stay focused on their task and feel that they are maintaining control of the system.

5. Visual design *Aesthetic integrity[Apple-87], Creating Aesthetic appeal[IBM-92] (S), Aesthetics[MS-95], Provide natural shades and colours[OSF-93] (S), Good visual design[Sun-89] (S).*

Visual design addresses the layout and appearance of objects. Most of the guides are careful to point out that the emphasis should be clarity and simplicity as opposed to fancy visual effects. In this way, attention can be focused on the task rather than the interface, and the overall aesthetic appeal should make the experience more pleasurable.

6. Robustness *Forgiveness[Apple-87], Forgiveness[MS-95], Anticipate errors[OSF-93] (S), Use explicit destruction[OSF-93] (S).*

A common contention among most style guides is that users are prone to make at least some errors, in part because many users prefer to learn by trial-and-error rather than reading manuals. Thus, systems are expected to prevent errors by: (a) providing clear guidance to the user, (b) being flexible in their expectation of user behaviour, and (c) providing cues and warnings which alert the user to exercise caution. In the event that an undesirable action does occur, the software should make it easy to recover.

7. Efficiency *Accommodating users with different levels of skill[IBM-92] (S), Making the user interface transparent[IBM-92] (S), Meaningful classes/objects[IBM-92] (S), Keep interfaces natural[OSF-93], Progressive disclosure[OSF-93], Efficiency[Sun-89], Keyboard equivalents and accelerators[Sun-89] (S), Pop-up windows and menus[Sun-89] (S), Progressive disclosure[Sun-89] (S).*

Efficiency measures how quickly and accurately a user can achieve the tasks they are using the system for. According to the guides, this is typically achieved by understanding the tasks users must perform and designing the interface from there. The aim is to make the interface "transparent", or "natural", so that the user can focus on their tasks rather than the interface itself. Thus, efficiency is supported by principles such as consistency, simplicity, familiarity, and visual design. Furthermore, this principle asserts that the

system should assist users in attaining proficiency. Flexibility is therefore important, so that the user can customise the application and control it in ways which would be too complicated for novices. Systems must also ensure that experienced users are aware of advanced features, without making the interface too complicated for novices. One recommendation most style guides make is progressive disclosure, i.e. providing mechanisms to expose more advanced operations as the user becomes more experienced.

Appendix B

Case Studies for Safety-Usability Patterns

B.1 Introduction

This Appendix provides summaries of the case studies that were particularly instructive in the formation of the safety-usability patterns, described in Section 7.4.

This is a replica of the case studies which were provided to subjects in the empirical study described in Section 7.5.

B.2 Oil Pressure System

Fields and Wright [79] describe a aircraft hydraulic monitoring system. The system consists of two dials which show the current hydraulic fluid level in each of two reservoirs and two switches (one for each reservoir) which indicate which of two control surfaces the reservoir currently supplies. A reservoir can supply both rudder and aileron but a rudder or aileron can be supplied by only one reservoir. The design of the system is shown in Figure B.1.

When confronted with a loss of fluid from either reservoir, the pilot of the aircraft must select a setting of the switches that minimises fluid loss and simultaneously determine the parts of the system that are leaking. The structure of the system is represented in Figure B.2.

The display is represented informally as shown in Figure B.1. In the worst *correctable* case, there may be a leak in a servodyne of one colour and a reservoir. For example, both the blue rudder servodyne and also the blue reservoir may be leaking. To correct the leaks, the pilot must switch both control surfaces to the green reservoir.

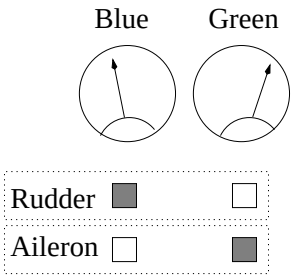


Figure B.1. Safety-critical case study: User-interface of hydraulic system (adapted from Fields and Wright, 1998 [79]).

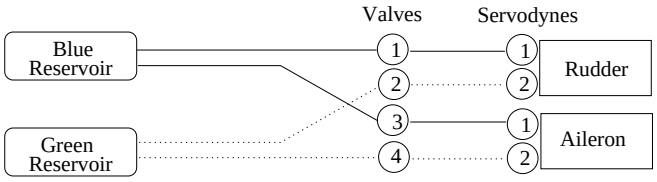


Figure B.2. Safety-critical case study: Physical hydraulics system environment (adapted from Fields and Wright, 1998 [79]).

B.3 Hypodermic Syringe

Dix [71, p.6] describes an automatic (computerised) syringe. The primary task engaged in by a user is to enter a dose for the syringe before applying the device to a patient. The original user-interface for the system has a calculator style interface that enables doses to be rapidly entered (see Figure B.3(a)). However, because the syringe could be injecting pharmaceuticals that are lethal outside a safe range, the original design does not sufficiently consider risk. When risk is taken account of, a better design is given in Figure B.3(b).

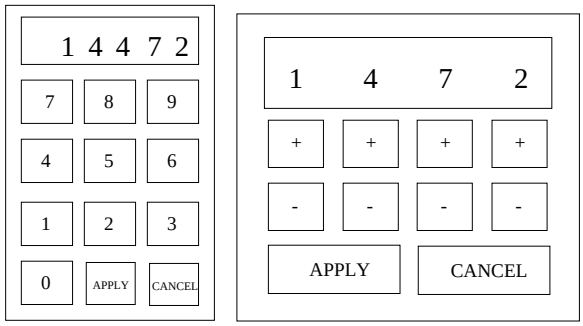


Figure B.3. Safety critical case study: Syringe design. (a) shows original design ; (b) is the revised design (adapted from Dix, 1998 [71, p.6]).

In the modified design, the user cannot enter doses as quickly and more effort is required to do so (so usability is reduced), but the system is safer because a single extra key press is less likely to produce an order of magnitude dose error. Additionally, the modified system provides error tolerance by allowing the dose to be changed (which also enhances usability).

B.4 Druide

The Druide system is a prototype Air-Traffic Control (ATC) system under development by the French aviation authority, CENA (Centre d'Études de la Navigation Aerienne). The prototype has formed the basis for a CHI (Computer-Human Interaction Conference) workshop on designing user-interfaces for safety-critical systems [183]. The analysis of Druide in this thesis is based on descriptions in [183].

A typical instantiation of the interface for Druide is shown in Figure Figure B.4.

The Druide ATC system is based on a data-link channel that is accessed from a menu-driven graphical user-interface, a radar screen annotated with flight information for each aircraft (call-sign, speed, heading and next beacon on its route) and paper strips that describe the flight plan for each aircraft. The paper strips are originally produced from flight plans submitted by the airlines. A controller is responsible for a sector of airspace including maintaining separation between aircraft. When changes are made to the flight plan for

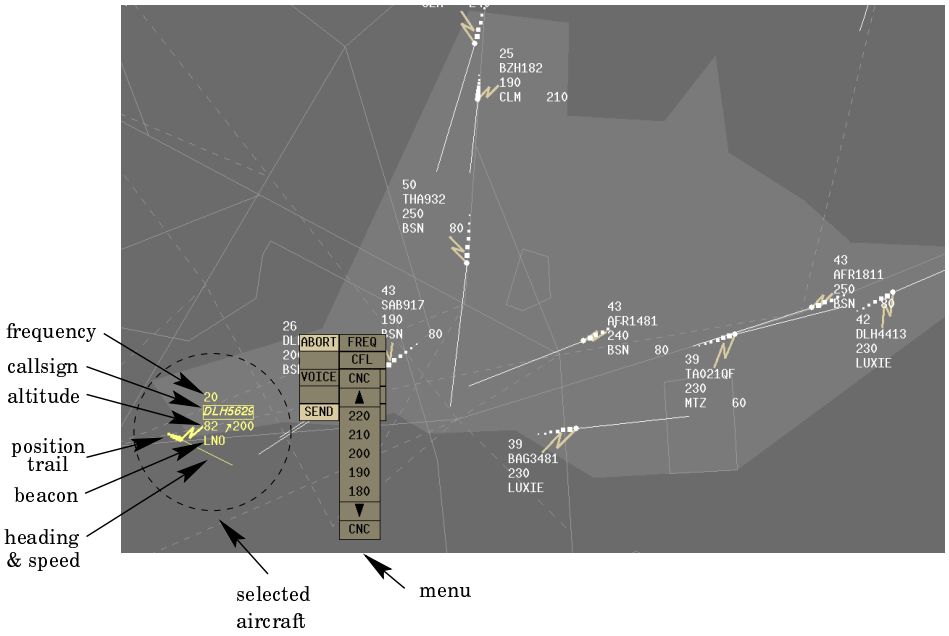


Figure B.4. Safety-critical case study: Druide radar display (adapted from Palanque et al., 1998[183]). Aircraft selected by the user have been manually circled for publication here .

an aircraft in a sector, the changes are recorded on the corresponding paper strip. When an aircraft enters a sector, its pilot must communicate with the controller. The controller manages that aircraft while it is in their sector before "shooting" the aircraft to a new sector (notifying the next controller of the handover of control). Managing an aircraft involves communicating via radar view and manipulating the paper strips. The controller may request that an aircraft change altitude, beacon, frequency, heading or speed.

In Figure B.4, the user is shown communicating a new altitude (CFL or Current Flight Level) to an aircraft. The selected aircraft with which the user is communicating is displayed in a distinguishable colour; selecting an aircraft produces a pop-up menu. When the user clicks on the "CFL" entry in the menu, a further menu of altitude possibilities appears. The user selects from this menu and then clicks the "SEND" button (alternatively, if a mistake has been made, the user may click "ABORT").

B.5 Railway Control

The Railway Control system is an Australian system, currently under development, for controlling the movement of trains on inland freight and passenger lines, preventing collisions between trains and maximising usage. The system is similar to an ATC system (with rail-traffic controllers and train drivers in lieu of air-traffic controllers and pilots) but is concerned with train movement, for which traffic movement and hence collision risk is more restricted. Figure B.5 shows the controllers screen for the system.

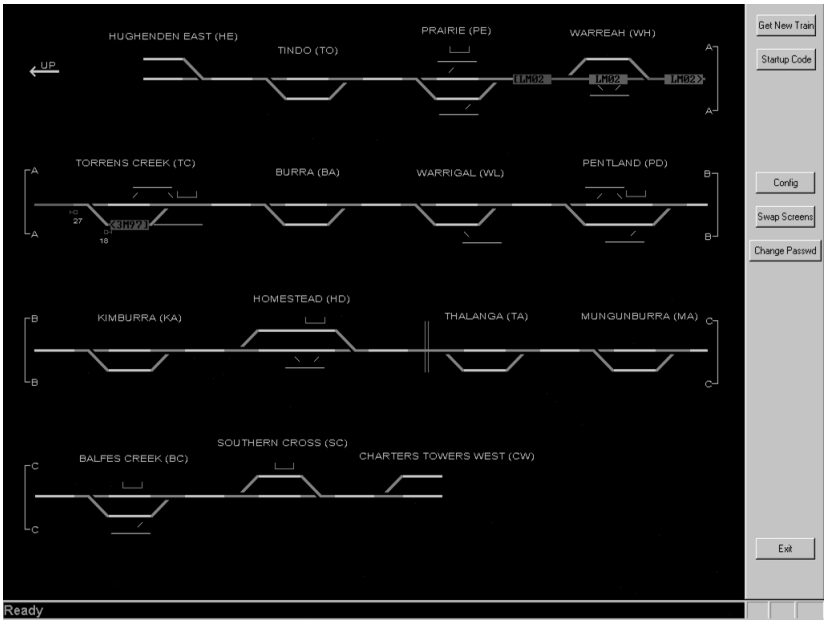


Figure B.5. Safety-critical case study: Controller's screen for Railway Control system

The controller and driver are in communication via a mobile phone or radio. For example, controllers may request drivers to move their trains from one location to another. When a controller issues such a

command to a driver the sequence is as follows:

1. controller (command)→ driver
2. controller ←(confirm) driver
3. controller (reissue)→ driver

The first step sends the command in coded form. The second step sends it back in coded form using a simple transformation of the original message. The third step sends auxiliary information that is compared against the command sent in the first step as a double-check of correctness. In each case, the recipient of a message types the code that is received into an entry field on their display. If all three steps are passed, the driver and controller will be presented with a text version of the command which is checked by the driver reading the text to the controller and the controller confirming that the command received is correct. Finally, if agreement is reached that the message has been correctly received, the driver commences execution of the command (e.g., by moving their train to a new location).

Commands to move a train to a new location are formulated by the train controller double-clicking on the train that is the subject of the command and then on the location that the train is to move to (see Figure B.5). In Figure B.5, the train is at Torrens Creek station and has been cleared through to just after Warreah station. Once a train has been instructed to move to a new location, all the track that the train must occupy in the course of executing that command becomes unavailable for use by any other trains. As the train moves along the track, the driver of the train contacts the controller to release sections of the track, so that other trains may move onto that track.

Appendix C

Descriptions of All Safety-Usability Patterns

C.1 Introduction

This appendix provides the full text for all patterns in the safety-usability language, which was summarised in Section 7.4.

These are the same patterns that were provided to subjects in the empirical study described in Section 7.5.

C.2 Task Management Patterns

Pattern C-7. Recover

Context: A task has been designed which could lead to a potentially hazardous state and it is possible to recover to a safe state from that potentially hazardous state. Risk can be sufficiently reduced by providing recovery paths for users, rather than reducing error likelihood. Prevention is costly or not possible.

Problem: How can we reduce the likelihood of accidents arising from hazardous states?

Forces:
- Hazardous states exist for all safety-critical systems; it is often too complex and costly to trap every state by modelling all system states and user tasks.
- Risk can be effectively reduced by reducing the consequence of error rather than its likelihood.
- When a hazardous state follows a non-hazardous state, it may be possible to return to a non-hazardous state by applying some kind of recovery operation.

Solution: **Enable users to recover from hazardous actions they have performed.** Recovering a task is similar to undoing it, but promises to return the system to a state that is essentially identical to the one prior to the incorrect action. In many safety-critical systems, this is impossible. If a pilot switched to a new hydraulic reservoir five minutes ago, then it is impossible to undo a loss of fluid in the meantime if the associated servodyne for that reservoir is leaking (see Piping Figure). However, it may be useful to provide a Recover operation giving a fast, reliable, mechanism to return to the initial reservoir. Recovering a task undoes as much of the task as is necessary (and possible) to return the system to a safe state.

This function can be assisted by:

1. helping users to anticipate effects of their actions, so that errors are avoided in the first place.
2. helping users to notice when they have made an error (provide feedback about actions and the state of the system).
3. providing time to recover from errors.
4. providing feedback once the recovery has taken place.

If this solution is not feasible, a more extreme way to deal with unwanted system states is to perform a `Shutdown`.

he system),

Examples: The Druide system (See Case Studies) implements message-sending via a `Transaction` mechanism. Air-traffic controllers can compose a message to an aircraft but then cancel it without sending (see Figure). In the Hypodermic Syringe case study, users can recover from an error by using the +/- button to alter the entered value to a safe value.

Resulting Context: After applying this pattern, it should be possible for users to recover from some of their hazardous actions. The `StepLadder` pattern facilitates recovery by breaking tasks into substeps, each of which may be more easily recovered than the original task. The user should be informed of what the previous state is that the system will revert to, hence `Trend` may help users execute `Recover`.

Pattern C-8. Stepladder

Context: The system is defined by a set of tasks that are decomposed into logically simpler tasks and the effect/consequence of misperforming a task cannot be readily diminished.

Problem: **How can we guide the user through complex tasks?**

Forces:
- It is desirable for the user to remain familiar with low-level tasks, so they are capable of dealing with novel situations.

- When performing a complex task, it is easy for users to forget what they have and have not done. This is especially true when there are other distractions.
- Users may eventually see the task sequence as a single, aggregate, task.

Solution: Identify complex tasks and explicitly split them into sequences of simpler tasks. In some cases, the task sequence may form a new task itself, for example a Wizard in MS-Windows is considered a separate task which enables several smaller tasks to be performed in a sequence. In other cases, there is no explicit representation; it is simply a design consideration which has led to the creation of several individual tasks. Even in this case, though, the user's tasks may be controlled by the system's mode, and `Behaviour Constraints` and `Affordance` can be applied to help the user identify what task comes next.

Examples: The concept of explicit procedures is well-established in safety-critical systems design. Aircraft crew are provided with reference cards explaining step-by-step what to do in emergencies [79]. Sometimes, controls on machines are arranged in a way which suggests a certain execution order. As an example from the non-safety-critical domain, Melbourne's public transport ticket machines require the user to specify three variables (zone, concession, expiry time), place money in the machine, and then collect the ticket. Even though the three variables can be entered in any order, the design of the machine suggests a particular order, arbitrary though it may be. The overall left-to-right ordering of controls provides a `Affordance` as to the appropriate sequence and suggests to users what to do next.

In the Hypodermic Syringe case study, in the usual case, several simpler actions are required to equate to the corresponding action when using a keypad. The positioning of the +/- buttons affords the appropriate sequence.

Resulting Context: After applying this pattern, the user should have an improved idea of what tasks need to be performed at any given time. The pattern works best in tandem with `Transaction`. Each few rungs of the stepladder forms a `Transaction`. This way, many of the individual tasks will be easily recovered from, because they will not be conveyed immediately to the broader system. By splitting the original task into sub-tasks, the consequence of each step may be less than for the original task and `Recover` may become easier to apply.

The stepladder can be used to structure `Affordances`.

Pattern C-9. Task Conjunction

Context: A task has been designed which has a relatively high risk of producing a hazardous state and error cannot be prevented, e.g., because the task involves data-entry. The task is not time critical.

Problem: How can we check whether a user's action matches their intention?

Forces:
- Redundancy is widely used in the safety industry to avoid hazards arising due to a component of the system failing. The system is said to have no single point of failure.
- Entry fields, or screens in a user-interface can be regarded as components of the system.
- Operators are likely to make errors on simple or repetitive tasks such as data entry.

Solution: **Reduce errors by requiring that the user perform tasks multiply.** The user's performances on each iteration of the task are compared and the outcome used to determine whether the task has been correctly performed.

Redundant tasks are an error detection technique. Redundancy reduces the efficiency with which users can perform tasks and therefore the "raw" usability of the system, but often enhances system safety by enabling error detection. Another variant is requiring the same task to be performed by two different users, as in a detonator which can only be activated by two people.

The `Task Conjunction` pattern is similar to `Transaction`, in that it requires several actions before any commitment is made. However, the intention differs. In `Task Conjunction`, there is only one fundamental change, but it is subject to verification actions. In `Transaction`, each action provides *new information* about desired changes.

The conjunction must not be able to be circumvented, e.g., on the Therac-25 machine, users could press "Enter" to confirm a value rather than re-type the value. Pressing "Enter" soon became automatic for the users [134].

Examples: Redundancy in software has a long history. Communication protocols, for example, use check-bits and other data to enable error detection and/or correction.

The Railway Control system requires that codes be exchanged between controllers and train drivers three times before a command is issued. The redundancy helps reduce the likelihood of an inconsistency between the controller's perception of the command that is issued and the driver's perception of the command.

Resulting Context: The original task is redefined as a conjunction of redundant sub-tasks, to which `Transaction` may be applied.

Pattern C-10. Transaction

Context: Actions are not time-critical and hence can be "stored-up" before being applied and:

- Sub-steps can be undone.
- The effect of the task as a whole is difficult to undo.
- Risk is relatively low compared to cost and difficulty of prevention.

Problem: For many real-time systems, it is difficult to provide a `Recover` action which has any practical value, because the system usually changes irreversibly by the time the user tries to recover from the unwanted action.

How can we improve recoverability?

Forces:
- It is relatively easy to recover tasks that do not impact on real-world objects.
- Often, reversal is useful to iteratively define a task's parameters.
- Transactions are used in data-processing to enable the effect of a sequence of actions to be "rolled-back" to the state at the commencement of the transaction.
- Transactions bundle a sequence of task steps into a single task, hence they are ideal for structuring interaction in terms of overall goals and sub-tasks.

Solution: **Bundle several related task steps into a transaction, such that the effect of each step is not realised until all the task steps are completed and the user commits the transaction**. By grouping task steps in this way, it becomes very easy to `Recover` the effect of a sub-step before the transaction as a whole has been committed. In addition, because errors are deferred until the transaction is committed, users have more time to consider their actions and to recover from them if appropriate.

Each transaction should involve task executions and information that is physically grouped on the users console or display. For example, a data entry transaction might be implemented as a pop-up window with commit and abort buttons to either accept or reject the information entered by the user.

Examples: In the Druide system, messages to aircraft are first constructed, and then sent to the aircraft using the "SUBMIT" button. The user can cancel a message before they submit it. A similar facility is available in some email systems which actually hold mail for several minutes before sending it.

A standard dialogue box supports this pattern. The user can easily enter data via text fields and buttons, but none of these choices matter until they hit a confirmation button (e.g., one labeled "OK").

Resulting Context: Task steps are grouped into transactions with commit and abort options for each group. The commit step in a transaction can quickly become automatic for the skilled user. If it is appropriate for the transaction's sub-tasks to be constructed iteratively, then the transaction can be viewed as a form of `Stepladder`. A sequence of transactions themselves can also form a `Stepladder`.

C.3 Task Execution Patterns

Pattern C-11. Affordance

Context: A task has a limited range of valid ways in which it can be performed and failure to perform the task correctly has hazardous consequences. For many systems, it is possible for the user to perform a variety of physical actions at each point in performing the task; not all will produce the required actions for the task.

Problem: How can we enhance assurance that the physical actions performed by users produce the desired effect?

Forces:
- It is possible for the user to have the right intentions, but perform the wrong action due to a slip.
- Slips can be reduced by providing appropriate affordances [177].

Solution: Provide cues to an operator that enhance the likelihood that the operator will perform the physical actions appropriate to performing a particular action.

The cues are effectively memory aids that remind the user to perform an action by a particular sequence of executions, avoiding slips. Such cues include distinctive executions for distinct actions and distinctive identifiers for distinct objects that the user can manipulate. The actions performed are matched to the outcomes of the actions and the user's physical expectations and capabilities. In addition, the same execution should not have several different outcomes according to the model or type of equipment that the user is using. Incorrect executions should clearly indicate that the operation has not been performed successfully.

Examples: Norman [177] gives several examples of doors that provide affordances, the physical characteristics of the door indicate the way in which the door should be used: for example a door knob may indicate by its shape that it should be twisted, whereas a flat bar on a door indicates that the door should simply be pushed. Failure to perform the operation of opening the door is indicated clearly because the door remains shut. Similarly, the landing gear on an aircraft is operated by a lever that is pulled downwards, mimicking the effect of the operation on the position of the aircraft's undercarriage.

Graphical toolkit components such as tabbed dialogues afford correct action by the user, because they mimic "real-world" entities that offer particular operations and indicate the availability of those operations by physical cues. Similarly, data blocks in Druide afford clicking because that is the characteristic operation to be applied to regions of marked text in a graphical display.

Design Issues: Selection of cues depends on the user's perceptual limitations. A user may not notice small differences in shape if they are occupied by other activities. General user characteristics must also

be considered, e.g., distinguishing between red and green might not be appropriate if colour-blind males interact with the system. Customisation of the interface may be necessary to accommodate the needs of all users.

Resulting Context: Certain types of user error ("slips" as described by Norman [177]) are less likely. `Warning` should be used to notify the user if an operation has not succeeded. There should be a `Reality Mapping` to provide the user with the state of the system, so they can determine whether the operation has occurred correctly. Because error may occur, `Recover` should be applied to enable the effects of error to be recovered from where possible. For those errors for which risk is too high for this pattern to be applied, `Behaviour Constraint` or `Interlock` should be considered. If affordance involves use of toolkit components, `Separation` should be considered. A `Preview` is a simple way of affording correct actions by showing what the outcome of an action will be.

Pattern C-12. Separation

Context: The system provides several actions for which the corresponding physical executions are very similar. Alternatively, several information displays are provided for which the layout and presentation are very similar. When one is appropriate, the other is not and possibly vice versa. In addition, it is not possible to predict hazards and prevent the potentially hazardous action.

Problem: A system is constructed from components that limit the scope for distinct executions and presentations (e.g., a graphical toolkit), so that the potential for confusion between components in different contexts arises.

How can we reduce the likelihood that users will inadvertently perform the wrong action or misinterpret information?

Forces:

- We would like to reuse components because modern systems usually incorporate graphical interfaces and it is impracticable to not use them where safety will not be compromised.
- Even if a toolkit is custom-built, the widgets and interaction mechanisms must then be reused in the design.
- Systems are always built within a budget.
- When widgets are reused, unless the customisation is extensive, components will often appear similar to users.
- Reusing commercial components means we cannot as easily customise them.
- The potential for user error increases as the similarity and proximity of controls increases.

Solution: Separate two controls (physically or logically) if they are operated in a similar way.

Examples: In the Druide system, the pop-up menu separates the buttons corresponding to the "SEND" and "ABORT" actions. Most style guides recommend separation for distinct operations that are accessed by similar controls (e.g., most Windows programs separate the "OK" and "Cancel" buttons in dialogues [165]).

Resulting Context: System components that are operated by similar physical actions are physically or logically separated. `Affordance` and `Distinct Executions` may also be used to reduce operator execution errors.

Pattern C-13. Distinct Interaction

Context: Two or more tasks are performed by a similar sequence of physical actions, and confusion between the tasks (e.g., by performing the terminating steps of task B following the commencing steps of task A) may result in a hazard. In addition, it is not possible to predict hazards and prevent the potentially hazardous action at the point at which it might be erroneously selected.

Problem: How can we reduce the likelihood that users will confuse similar tasks?

Forces:
- Reuse of graphical toolkit components enhances consistency and makes a user-interface easier to learn.
- Reuse of such components also increases the consistency of the operations required to perform tasks, so that distinct tasks may be accessed by similar physical executions.
- Tasks that have similar execution sequences are likely to be confused by users.
- Users confuse tasks because of memory failures.

Solution: Distinct actions that can be confused, leading to hazardous consequences, should be accessed by distinct physical interactions. However, distinct physical interactions reduce reuse, making the system harder to learn. Training and memory aids can help overcome errors arising from users not remembering the correct execution to perform a task.

Examples: The Hypodermic Syringe system uses alignment of +/- buttons with the corresponding display digit to reduce motor errors in which a wrong button is pressed.

Resulting Context: Controls that might be confused by the operator are operated by distinct physical interactions. The interactions required to operate a control should be afforded by the control (see `Affordance`). `Separation` and `Preview` are alternative solutions.

Pattern C-14. Behaviour Constraint

Context: It is possible to determine system states where certain actions would lead to an error.

Problem: The most direct solution to improving assurance through diminished user error is to prevent users from making errors.

How can we prevent users from requesting undesirable actions?

Forces:
- Even if we provide appropriate information and guide users by providing `Affordances`, users will inevitably make some slips and mistakes.
- The system has a large amount of data available concerning the state of objects.
- In some circumstances, designers will be aware that certain tasks will be undesirable in certain situations.
- It is preferable to trap user errors before they impact on the system, rather than detecting potentially hazardous system states and then attempting to rectify the situation.

Solution: **For any given system state, anticipate erroneous actions and disallow the user from performing them.** The idea here is to prevent the action from occurring in the first place, rather than dealing with whatever the user does. The logic could be programmed directly, e.g., "If the plane is in mid-air, disable Landing Gear button". It could also be implemented via some intelligent reasoning on the machine's behalf, which would require the machine to understand what tasks do and where they are appropriate.

This is a very fragile approach which should be used with extreme caution. It assumes that the condition is measured accurately and that ignoring the constraints always implies a worse hazard than following them. It is therefore risky in unforeseen circumstances. This can be somewhat alleviated by a mechanism allowing an appropriate authority to override the constraint.

A further disadvantage is that this pattern leads to a less flexible user-interface (i.e., the interface is moded) and the user may become frustrated with the structured form of interaction if they do not properly understand the tasks they are performing. In a safety-critical domain, this should be a less severe problem than normal, because the user should be highly trained.

Behaviour constraints are usually implemented as an additional system mode, e.g., by "greying out" menu items or buttons where appropriate. As such, behaviour constraints require close automatic monitoring of system state and therefore a frequently used partner pattern is `Automation`.

Examples: Kletz [126] gives everyday examples, such as components that will only fit together in one way, making incorrect assembly impossible. Norman [177] also gives such everyday examples, such as fire stairs that have a rail blocking access to the basement stairs; the error of proceeding all the way to the basement is prevented. In the Druide system, users are only given access to menus to change the system state, preventing errors associated with entering invalid data. In the Railway Control system, users cannot direct a train to move to an occupied section of track.

Resulting Context: Some user errors are no longer possible. The system will usually be more heavily moded than prior to applying the pattern. The system may inform the user that a requested operation is unavailable (i.e., applying the `Warning` pattern).

C.4 Information Patterns

Pattern C-15. Abstract Mapping

Context: The system enables users to interact with complex real-world objects.

Problem: Modern computer-based systems may have extremely complex internal state. To operate the system, the user needs to be able to understand what the state of the system is. **How can we reveal to the user what they need to know about the state of the system without swamping them with information?**

Forces:
- Humans have a limited capacity for processing information.
- For many safety-critical systems, the amount of information that is relevant to the users operation of the system exceeds the processing capacity of the user.
- Many decisions that the user must make in ensuring safe operation of the system depend on the overall trend of the system state, rather than precise details.
- Many systems allow some margin for error so that precise representation of the system state is not necessary for safe operation.

Solution: Provide an abstract representation of complex information so that it is comprehensible by the user. Abstract representations can be used when it is not helpful to directly represent the real-world objects. One benefit of computers is that we can construct summaries—indirect mappings—which aid human decision-making. Users should not have to perform complicated mental operations to assess the margins of system operation [166]. More abstract representations can be used in situations where we are prevented from showing real-world objects. For instance, bandwidth might be constrained. Displaying unnecessary parameters increases system load but redundant parameters can be used to automatically check the consistency of information which can then be displayed as a single value [166].

Examples: The Druide interface provides the plane's speed, an abstract mapping. This is a variable which could be derived from the reality mapping, but is shown explicitly to the user, because a user's estimate would lack accuracy, consume time, and distract them from other tasks.

Resulting Context: Information will be displayed to the operator using abstract representations unless a closer `Reality Mapping` is feasible.

Pattern C-16. Redundant Information

Context: The user is required to perceive and interpret information and act on that information in maintaining the safe function of the system. The information is complex, or could be misperceived or misinterpreted.

Problem: How can we enhance the likelihood that the user correctly perceives and interprets safety-critical information?

Forces:
- Some information is more salient than others.
- Safety-critical systems may be complex, with large amounts of information that needs to be available to the user.
- The amount of display space available is limited.
- Providing too much information will swamp the user; the information that is displayed needs to be chosen carefully.

Solution: Provide more than one way of viewing the information, so that the likelihood that it is misunderstood or ignored is lessened. Redundancy in user-interface designs can be viewed as an extension of the usual safety-critical viewpoints advocating redundant hardware and "n-version programming" (but note that such approaches are often unsuccessful in software design). The field of communication studies has looked at situations such as combination of audio and visual cues in television (e.g., both auditory and visual warnings).

Examples: The use of the international phonetic alphabet ("Alfa, Bravo, Charlie ..." instead of "A, B, C ...") [126] is a form of redundant input that has been in use for many years to improve human-human communication. Similarly, the purpose of levers can be more readily comprehended if there are adequate cues beyond just a label, e.g., shape, position, colour (see [126]).

Resulting Context: Information that is vital to safety will be displayed to the operator in several redundant forms. `Redundant Information` will usually take the form of an `Abstract Mapping`.

Pattern C-17. Reality Mapping

Context: The system enables users to interact with real-world objects.

Problem: Digital systems usually create a degree of separation between users and physical objects. In some aircraft, for example, the visual feedback is entirely virtual. Benefits notwithstanding, there is a risk that vital information may be unavailable.

How can the user check that objects under their own control, or under the system's control, are aligned with their expectations?

Forces:
- Information concerning the current state of a safety-critical system needs to be displayed clearly and succinctly, so that the user can quickly ascertain whether a hazard has arisen or could arise and can take appropriate action.
- When parts of the system have been automated, there is a temptation to assume that the user does not need to see the state of certain objects. However, failures do arise, and human intervention is often necessary[175]. Information about the environment is necessary to help humans monitor the system, and, if necessary, intervene.

Solution: Provide a close mapping to reality where possible and supplement it with abstract representations. To help the user build an accurate model of the domain, it is important where feasible to maintain a close mapping between physical objects and their virtual counterparts. A close mapping to reality will help the user observe and diagnose problems effectively. The mapping should incorporate the necessary information that the user can ensure safe operation.

Examples: The Druide system provides an accurate, directly manipulable, display of the airspace and the aircraft in it. The Oil Pressure system provides analogue displays of oil pressure and uses proximity of components to convey relationships.

Design Issues: When mapping to reality, the appropriate level of detail will be guided by knowledge of the user's characteristics and tasks. Object-orientation provides a good way to reduce semantic distance (distance between the human's model and the real world) because each object in the display for the system represents a corresponding object in the task domain. Analogue displays also reduce semantic distance because they enable direct comparison of magnitudes (e.g., [166]). Similarly, minimise articulator distance so that physical actions mimic their meanings.

In situations when the representation is complex, abstract representations can be used to extract from reality any information which is likely to help users in their task.

Resulting Context: The result is a mapping from reality into suitable display objects. Since the display will not be optimal for all cases, the `Interrogation` pattern can be used to help the user refine the information provided. An `Abstract Mapping` may be used when reality mapping of the system state is not necessary for safety, or is infeasible.

Pattern C-18. Trend

Context: Users need to formulate and follow task plans that involve attention to the change in state of the system, e.g., where an action must occur if the state is in a certain configuration, or when a state change occurs.

Problem: How can the user be notified that the state has changed (i.e., the trend of the system is towards a hazardous state)?

Forces:
- Many user errors stem from memory limitations. Users may not notice that the state of the system has changed and that they should take action;
- Memory-based errors may occur even when the user has previously formulated a plan to perform a particular action when the state of the system reaches a particular configuration. For example, in air-traffic control, the user may need to change the altitude of an aircraft before it reaches a particular waypoint but may not be able to do so immediately because of other more pressing concerns; a hazard arises when the user fails to return to the original aircraft and change its altitude, after resolving the immediate concern.

Solution: **Allow the user to compare and contrast the current state with previous states.** This will help users assess the current situation and formulate plans for the future.

Examples: The Druide system displays aircraft as a trail of locations, with the most prominent location displayed being the immediate location (see Figure). The Oil Pressure system displays oil pressures in the left and right ailerons and a shaded region that indicates the oil pressure in the previous 5 minute interval (see Figure).

Design Issues: One common technique is to overlay previous states on the same display, showing the previous state in a less visually striking manner (e.g., muted colours). This is particularly effective for motion, as a trail of past motion can be formed. If this technique causes too much clutter, a replica of the past state can be placed alongside the current state. This, however, occupies valuable screen real estate, and may hamper direct comparison.

Resulting Context: State changes are explicitly displayed to the user. Display of the state change is a `Reality Mapping`. The change in state may also be brought to the users attention via a `Warning`, if it has a readily identifiable safety implication.

Pattern C-19. Interrogation

Context: The system is complex, with much information of potential use to the user in performing their work and not all attributes can be displayed at one time.

Problem: Most safety-critical interactive systems display to the user a representation of the system state. For many such systems, the state of the system is complex and cannot be represented in a comprehensible way. For such systems, displaying the entire system state at one time may obscure the most important components of the state and represent a potential source of user error.

How can the user have access to the entire state of the system without being overwhelmed by information?

Forces:
- Some of the information is more salient, or more often necessary than other components of the information.
- Some of the information is more readily displayed than other components of the information.
- Users have limited attentional capacity. For example, Halford [100] has shown that an upper limit of about 4 items can be processed in parallel.
- Display devices have limited resolution and capacity and can quickly become cluttered with information. Users have difficulty locating specific features on cluttered screens and this is particularly problematic when urgent information or control is desired.
- Designers cannot realistically envisage every possible information requirement.

Solution: **Provide ways for the user to request additional information.** This way, not all information needs to be shown at once.

If the user is monitoring automatic systems, provide independent information on the state of the system in case instrumentation fails; instrumentation meant to help users deal with a malfunction should not be able to be disabled by the malfunction itself.

The capability to interrogate should be obvious or intuitive; menus at the top of the screen or window are preferable to pop-up menus or mouse actions performed on the object of interest. The results of the interrogation should be able to be saved and reviewed at a later time.

Examples: The Druide system displays aircraft locations, relative speed, direction of travel, position trail, radio frequency, call-sign, altitude and beacon. Aircraft have many other attributes such as their flight plan, which are not displayed at all times because the result would produce unmanageable display complexity. Instead, controllers may query aircraft for such additional information, when it is needed.

Resulting Context: The mapping of system state to the display is now a mapping of only part of the state, the remainder of the state is hidden. The result is more efficient use of the display.

Pattern C-20. Memory Aid

Context: The task being performed by a user enables arbitrary interleaving of actions, with the possibility that an action will be forgotten.

Problem: Some safety-critical systems, such as air-traffic control, require the user to perform several tasks concurrently, with interleaving of task actions. In such systems, the potential is much greater than in non-interleaved systems for actions to be forgotten, leading to hazards arising.

How can users reliably perform interleaved tasks?

Forces:
- Users must remember to finish all tasks, including interrupted tasks.
- For some systems, the user must not inadvertently perform a task step more often than is required (for some systems and steps, a hazard may result).

Solution: **Provide ways to record the completion status of steps.** This will help the user to recommence later on without omitting or repeating tasks. Such memory aids may be either components of the computer system itself, or adjuncts to the computer system. Memory aids may be proactive, cuing the user to perform an action at a particular point in time, or when the system reaches a particular state; such memory aids may be warnings.

Examples: The Druide system uses paper strips to record flight details and the instructions that have been given to aircraft; such paper strips are commonly used in air-traffic control systems. The paper strips provide context for the controllers, they enable controllers to determine whether an aircraft is currently behaving in accordance with the instructions that have previously been issued and they enable the last action performed for a particular aircraft to be recorded. However because paper strips are an adjunct to the computerised ATC system, they cannot *actively* cue the user to perform an action at a particular point in time, or when the system reaches a particular state. Fields [79] describe checklists as a simple memory aid to ensure that all the steps in a safety-critical procedure are completed (e.g., piloting an aircraft).

Design Issues: Proactive memory aids may be set by the user or by the system. However system initiated warnings require that the system be aware that a user task has not been completed. Memory aids should cue the user with an urgency corresponding to that set by the user or in the case of system initiated warnings, with an urgency corresponding to risk. Passive memory aids should be visible to the user at all times, e.g., tags associated with an object in the display.

Resulting Context: The user is provided with active and passive memory aids. Passive memory aids may require `Reality Mapping`. Active memory aids may use `Warning` to notify the user that a condition (user or system defined) has been reached (and that the user should take appropriate action). Trend displays are a form of passive memory aid (see pattern `Trend`).

C.5 Machine Control Patterns

Pattern C-21. Interlock

Context: Risk is sufficiently high that measures to block the occurrence of error do not give sufficient assurance and the bounds of acceptable system outputs can be defined.

Problem: **How can we be sure that errors cannot lead to high risk hazards, even if they occur?**

Forces:
- Measures to diminish user errors are not necessarily sufficient assurance if risk is sufficiently high.
- `Behavioural Constraints` may not prevent all incorrect commands because systems are too complex to predict all possible states and events.
- Measures to diminish user errors should not necessarily be regarded as sufficient evidence of system safety and additional evidence may be required if risk is sufficiently high.

Solution: **Anticipate errors and place interlocks in the system to detect and block the hazards that would otherwise arise.** Interlocks can be embodied in hardware or software, preferably both. But there is no point creating an interlock if the system failure causes the interlock itself to work incorrectly.

Examples: Many modern motor cars come equipped with Anti-Lock Braking Systems (ABS). Such systems are interlocks, screening aberrant driver behaviour. If the driver presses too hard on the brake pedal, the ABS will override the driver's actions and maintain brake pressure at an optimum rate. The Therac-20 and Therac-25 machines are medical equipment designed to administer radiation doses to patients [133]. The Therac-20 machine has a hardware interlock that prevents extreme doses of radiation but the Therac-25 machine does not. The Therac-25 machine was involved in several well-publicised accidents that arose because of an error in the software which failed to correctly update the dose after the user had corrected it on screen. Detection of errors assumes that the hazard situation can be formulated in a straight-forward and error-free way. As an example in which this was not so, consider the Warsaw accident in which A320 braking was prevented because braking logic required both wheels on the ground (e.g., see [133]). These issues are considered further in `Automation`.

Design Issues: If a system is designed using only interlocks to prevent hazards arising from user error, removal of the interlocks opens the system to the possibility of hazardous operation. Interlocks therefore should always be used with `Behaviour Constraint` and `Intended Action` to provide 'defence in depth'.

Pattern C-22. Automation

Context: Consider this pattern if performing a function involves danger to a user, performing the function requires exceptional skill, e.g., as when the response time is far shorter than a human can normally achieve, or performing the function requires tedious or repetitive work.

Problem: Many safety-critical processes, such as nuclear power generation or aircraft control, also require manipulation of a large number of parameters to keep the system within safe margins of operation. However humans are not very good at monitoring and controlling a large number of parameters in

real time. Machines are good at such monitoring and control but typically cannot detect and correct all possible abnormal component failures.

How can system parameters be reliably maintained within safety margins even in the presence of component failure?

Forces:
- Part of the operation of the system involves maintaining parameters within defined safety margins.
- Users become fatigued when monitoring parameters to ensure they stay within safety margins.
- Users need to remain informed of what the system is doing so they can intervene in the event of component failure.

Bainbridge [10] maintains that it is not possible for even a highly motivated user to maintain attention toward a source of information on which little happens for more than half an hour. Hence it is humanly impossible to carry out the basic monitoring function needed to detect unlikely abnormalities. In addition, the user will rarely be able to check in real-time the decisions made by the computer, instead relying on a meta-level analysis of the 'acceptability' of the computers actions. Such monitoring for abnormalities must therefore be done by the system itself and abnormalities brought to the user's attention via alarms.

Solution: Automate tasks which are either too difficult or too tedious for the user to perform. Mill suggests that a function should be automated if [166]:

- performing the function involves danger to a user [166];
- performing the function requires exceptional skill, e.g., as when the response time is far shorter than a human can normally achieve;
- performing the function requires tedious or repetitive work.

and should not be fully automated (i.e., a human should be included in the control loop with responsibility for decisions) if a decision must be made that:

- cannot be reduced to uncomplicated algorithms;
- involves fuzzy logic or qualitative evaluation;
- requires shape or pattern recognition.

Appropriate design of automatic systems should assume the existence of error, it should continually provide feedback, it should continually interact with users in an effective manner and it should allow for the worst situations possible [175].

Examples: Typical examples of automation are cruise control in cars and auto-pilots. Autopilots reduce pilot workload; without autopilots pilot fatigue would be a very significant hazard in flight. Further, some aircraft (e.g., military aircraft) could not be flown without automatic control of some aircraft functions. However experience from case studies of crashes indicates that there is not always enough feedback and that this can lead to accidents [175].

Design Issues: The automated system needs to provide continuous feedback on its behaviour in a non-obtrusive manner. Careful consideration is therefore required of what aspects of the controlled process are most salient to safety and to ensure prominence of their display. If the system has been given sufficient intelligence to determine that it is having difficulty controlling the process, then active seeking of user intervention is appropriate.

Resulting Context: Aspects of the system that are unsuited to human performance are automated. Automation of a system will usually require that the system also notify the user of failures (see `Warning`).

Pattern C-23. Shutdown

Context: Shutting down the system leads to a fail-safe state and the service provided by the system can be halted for at least a temporary period, and:

- failure can lead to a serious accident within a very short time;
- shutdown is simple and low cost;
- reliable automatic response is possible.

Problem: How can safety be assured if a serious hazard has occurred?

Forces:
- In the event of component failure, a safety-critical system may continue to function, with graceful degradation of service, however such a malfunctioning system is inherently less safe than a fully-functional system.
- Systems such as factory plant can often be shut down in a safe state.
- Systems where failure of service is a hazard usually cannot be shutdown, e.g., aircraft, Air-Traffic-Control services, missiles.

Solution: When shutdown is simple and inexpensive, and leads to a safe, low risk state, the straightforward solution is to shut down automatically.

When a system cannot be shut down automatically because of cost or complexity, then the system must instead be stabilised, either manually or automatically [10]. If a failure can lead to a serious accident within a very short time, without shutdown, then reliable automatic response is necessary, and if this is not possible, then the system should not be built if risk is unacceptable.

Examples: McDermid and Kelly [160] give an example of an industrial press that automatically moves to a safe failure state (i.e., press closed) when the sensors for press movement are inconsistent (i.e., and therefore mitigations against user error are not operative so that a hazard exists).

Design Issues: Shutdown should be annunciated by a warning to the user that the system is no longer operating.

Resulting Context: The system shuts down on detection of failure. If a system cannot be simply shutdown, then `Automation` might be used to stabilise the system and `Warning` is indicated to bring the failure to the user's attention.

Pattern C-24. Warning

Context: Identifiable safety margins exist so that likely hazards can be determined automatically and warnings raised.

Problem: Computers are better than humans at monitoring the environment to check if a certain set of conditions are true. This is because humans become fatigued performing a tedious task repetitively. Furthermore, their attention can be distracted by other tasks, which might lead to them missing a critical event, or, upon returning, forgetting what they were monitoring.

How can we be confident the user will notice new system conditions and take appropriate action?

Forces:
- Computers are good at monitoring changes in state.
- Computers are good at maintaining a steady state in the presence of minor external aberrations that would otherwise alter system state.
- Computers are not good at determining the implications of steady state changes and appropriate hazard recovery mechanisms.
- Although users should still be alert for failures, their workload can be lightened and the overall hazard response time decreased, if searching for failures is at least partially automated. This requires a mechanism to inform users when a failure has occurred.
- Conditions may be user defined.

Solution: Provide warning devices that are triggered when identified safety-critical margins are approached. The warnings provided by the system should be brief and simple. Spurious signals and alarms should be minimised and the number of alarms should be reduced to a minimum [166]. Users should have access to straightforward checks to distinguish hazards from faulty instruments. Safety critical alarms should be clearly distinguishable from routine alarms. The form of the alarm should indicate the degree of urgency and which condition is responsible. The user should be guided to the

disturbed part of the system and aided in the location of disturbed parameters in the affected system area [11]. In addition to warning the user if identifiable hazards may occur, the system also should inform the user when a significant change has taken place in the system state: see the `Trend` pattern. The absence of an alarm should not be actively presented to users [166]. Warnings should be provided well in advance of the point at which a serious accident is likely. Warnings should be regarded as supplementary information; a system should never be designed to operate by answering alarms [166].

Examples: The Oil Pressure system raises an alarm when the oil pressure in the current aileron falls below a threshold. The HALO system (discussed in [11]) uses logical expressions to generate a reduced number of alarms from a total array of alarm signals. HALO alarms are formed on the basis of a combination of signals in contrast to conventional systems where alarms arise from violation of a single parameter. The Druide system alerts the user when a separation conflict develops between two or more users.

Design Issues: For serious failures constituting actual hazards, warnings may be moded, requiring the user to acknowledge the warning before proceeding further. Such moded warnings are called alerts. Alerts will often be auditory because hearing is a primary sense that is detected automatically; auditory alerts are less prone to being ignored due to "tunnel vision" [184]. Patterson [184] has shown that only four to six different auditory alerts can be distinguished reliably and that careful attention must be paid to ensuring sounds do not conflict, making combinations of alerts difficult to distinguish. Auditory alerts should be loud and urgent initially, softer for an intervening period allowing user action and cancelling of the alert, then load again if no action has occurred after the intervening period.

Resulting Context: Hazards are identified and logic installed in the system to warn users when hazard margins are approached. Warnings should be replaced by `Automation` where possible. Warnings may be structured hierarchically, so that only primary failures that are responsible for a system or subsystem failure are displayed initially, with secondary failures appearing only at the user's request (i.e, applying the `Interrogation` pattern) [166]. If a warning is triggered, the user should have access to mechanisms for recovery (`Recover`) where possible.

Appendix D

Materials for Safety-Usability Patterns Experiment

D.1 Introduction

The following pages contain the materials that were provided to subjects during the safety-usability study described in Section 7.5. The materials are:

Study Overview Study overview for ethics purposes.

Consent Form Consent form for ethics purposes.

Instructions Instructions to subjects. Two versions exist, one for each group.

Design Tools List of principles provided to all subjects. In addition to principles, subjects in patterns group received list of patterns as in Appendix C, case studies as in Appendix B.1, and overview of patterns as in Section 7.4.1 (all were attached together).

Example Case Study Factory-based alarm system allowing monitoring of sensors.

Actual Case Study Application for radiologists who are performing laser emissions into the head.

Section numbering within the materials has been updated for publication here, to reflect location within the appendix.

D.2.1 Information Sheet provided to subjects prior to session

The University Of Melbourne

Computer Science and Software Engineering Department

Design for Human-Computer Interaction: Information Sheet

Investigators: Michael Mahemoff (moke@cs.mu.oz.au), Lorraine Johnston (ljj@cs.mu.oz.au).
Telephone: 9344-9100

This sheet is an overview of the study we are conducting.

Software usability is becoming an important issue for developers. Developers would like techniques which enable them to create software which meets users' needs, and lets users interact with software easily and efficiently. Thus, various researchers have proposed techniques which developers can use to improve usability.

We are considering possible approaches to designing for usability. "Design" in this context means considering how the user interacts with the system (as opposed to software design, e.g. object-oriented or Jackson Structured design). We would like to determine how useful the design approaches is, and see what kinds of designs people produce. Therefore, we will introduce you to the approach and ask you to apply it to a redesign problems. There are no right or wrong answers. We are evaluating how effectively the approaches can be used by a designer, rather than evaluating the designers themselves. At the end, the administrator will clarify your designs and ask you about how you found each technique.

It is anticipated that the entire session will take $2\frac{1}{2}$-$3\frac{1}{2}$ hours, and this will include a free pizza lunch. As required by the University's ethics guidelines, you are free to leave at any time and you may also withdraw your work at the same time. Not participating in this study, or choosing to leave early, will not affect your results or assessment in any way, and will not be discussed with anyone.

If you have any questions at this stage, please ask the administrator. You may also ask the administrator to clarify any information at any time during the session.

You may retain this sheet. To ensure that everyone has the same expectations going into this session, please do not show it to anyone, or discuss the study with anyone else who will be participating, for a period of four weeks. You may also wish to receive feedback on the overall study once it has been completed. To do this, either leave your email address on the consent form, or mail Michael Mahemoff (moke@cs.mu.oz.au) at any time until December 31, 2000.

D.3.1 Consent form provided to students at start of session

The University Of Melbourne

Computer Science and Software Engineering Department

Consent form for persons participating in Design for Human-Computer Interaction research project

Project: Design for Human-Computer Interaction.

Investigators: Michael Mahemoff (moke@cs.mu.oz.au), Lorraine Johnston (ljj@cs.mu.oz.au).

Name of participant: _____

1. I consent to participate in the above project.
2. I am aware that this project satisfies the University of Melbourne Research Ethics Committee guidelines for projects involving human subjects.
3. I have read the accompanying information sheet and I have had the opportunity to ask any questions about this study.
4. I authorise the investigators to use my results to assist them in the study.
5. I acknowledge that:
   (a) I am free to withdraw from the project at any time during this session, and if I do so, I may also withdraw any work I have performed during the session.
   (b) The project is for the purpose of research only, and is being used to evaluate design approaches, rather than to assess my own skills in any way.
   (c) The information I provide will be kept confidential, subject to any legal requirements, and my name will not be attached to the work I produce.
   (d) This consent form will be kept, among other consent forms, separate from the results produced by myself and others. In the extremely unlikely event that the investigators are asked to demonstrate that the results have not been fabricated, I may be asked to verify that I participated in this study.

6. I can provide my e-mail address if I wish to obtain feedback about the study once results have been analysed; otherwise, it is not required.

   E-mail Address (Optional): _____

   Alternatively, you can mail Michael Mahemoff (moke@cs.mu.oz.au) at any time between March 31 and December 31, 2000, to receive feedback.

Signature: _____ **Date:** ___ / ___ / _____

dd mm yyyy

Design for Human-Computer Interaction—January 2000

Michael Mahemoff and Lorraine Johnston

D.4.1 EXPERIMENT OVERVIEW

We are interested in techniques which help people produce more usable software. To help us evaluate some of our methods, we will introduce you to a techniques, show you some initial design ideas, and ask you to improve on the sketches using the techniques. Please remember that we are evaluating the ease and effectiveness of our techniques rather than assessing your own skills.

D.4.1.1 What is usability?

Usability is a quality associated with the interaction between users and computers. The ISO-9241 standard defines usability as follows:
Usability: The *effectiveness*, *efficiency* and *satisfaction* with which specified users achieve specified goals in particular environments [Italics added].

In this study, we are looking at a safety-critical systems. Therefore, we are especially interested in designing systems which prevent users from making mistakes and minimise the effects of those mistakes which do occur.

D.4.2 REDESIGNING FOR USABILITY

D.4.2.1 The materials we will provide

We are asking you to build on an existing design for a safety-critical system. We will tell you its purpose and provide you with some initial user-interface sketches.

D.4.2.2 How to redesign the system

You have been provided with a list of design principles. You are free to use the principles in any way you like. You might design directly from the principles, perhaps by stepping through the list and seeing which how each applies. In other situations, you might design using your own ideas and relating them back to the

principles. At times, you might also decide to design according to your own ideas, leaving the principles aside altogether.

D.4.2.3 The outputs we would like you to produce

We would like you to redesign the system to improve its usability, with a focus on safety. To clarify your final design, the administrator will ask about how the user would accomplish certain tasks, what they will see, etc. In particular, you can consider all of the following elements:

Functionality You can probably think of tasks the user might like to accomplish which are not possible in the initial prototype.

User-interface design You can redesign the user-interface as much as you like.

External elements You can discuss any other aspects as well. For instance, in the alarm case study, you might comment on the alrms themselves even though they are not directly controlled by the computer's interface.

The redesign will mostly be composed of user-interface sketches and annotations. You can also write down notes about aspects which have no visual component.

D.4.3 FINISHING UP

The administrator will clarify your work and enquire what you thought about the design approach. We will also be interested to see how the final design related to the design technique.

Thankyou for spending some time to help us with this study.

Design for Human-Computer Interaction—January 2000

Michael Mahemoff and Lorraine Johnston

D.5.1 EXPERIMENT OVERVIEW

We are interested in techniques which help people produce more usable software. To help us evaluate some of our methods, we will introduce you to a techniques, show you some initial design ideas, and ask you to improve on the sketches using the techniques. Please remember that we are evaluating the ease and effectiveness of our techniques rather than assessing your own skills.

D.5.1.1 What is usability?

Usability is a quality associated with the interaction between users and computers. The ISO-9241 standard defines usability as follows:
Usability: The *effectiveness*, *efficiency* and *satisfaction* with which specified users achieve specified goals in particular environments [Italics added].

In this study, we are looking at a safety-critical systems. Therefore, we are especially interested in designing systems which prevent users from making mistakes and minimise the effects of those mistakes which do occur.

D.5.2 REDESIGNING FOR USABILITY

D.5.2.1 The materials we will provide

We are asking you to build on an existing design for a safety-critical system. We will tell you its purpose and provide you with some initial user-interface sketches.

D.5.2.2 How to redesign the system

You have been provided with a list of principles and patterns. The patterns are intended to show you how the principles can be successfully applied, thus the patterns should be the main tool for redesign. You can also use the principles if you feel it's appropriate. You can also rely on your own ideas, aside from patterns

and principles, where appropriate.

You are free to use the patterns in any way you like. You might find the diagram and Resulting Context sections useful, to see which patterns lead to other patterns. In other situations, you might just jump from one pattern to another. You might also step through the list of patterns and see how each one applies.

You do not have to use the entire pattern. You might find it easier to look only at certain fields. Again, that is your choice.

D.5.2.3 The outputs we would like you to produce

We would like you to redesign the system to improve its usability, with a focus on safety. To clarify your final design, the administrator will ask about how the user would accomplish certain tasks, what they will see, etc. In particular, you can consider all of the following elements:

Functionality You can probably think of tasks the user might like to accomplish which are not possible in the initial prototype.

User-interface design You can redesign the user-interface as much as you like.

External elements You can discuss any other aspects as well. For instance, in the alarm case study, you might comment on the alrms themselves even though they are not directly controlled by the computer's interface.

The redesign will mostly be composed of user-interface sketches and annotations. You can also write down notes about aspects which have no visual component.

D.5.3 FINISHING UP

The administrator will clarify your work and enquire what you thought about the design approach. We will also be interested to see how the final design related to the design technique.

Thankyou for spending some time to help us with this study.

Design for Human-Computer Interaction—January 2000

Michael Mahemoff and Lorraine Johnston

Design principles are broken into two sections below:

Design Principles for Increasing Robustness These are closely related to safety-critical concerns, i.e. preventing errors, reducing their effects, and recovering from them.

Design Principles for General Usability These are typical usability concerns, e.g. user efficiency. These are secondary concerns in safety-critical systems, but should still be taken into account during design.

D.6.1 Design Principles for Increasing Robustness

D.6.1.1 Error Prevention

- Where possible, prevent the user from placing the system in an unsafe state. Ideally, unsafe states can be identified in advance and the system can be designed such that the unsafe state is impossible. Other error prevention mechanisms include hardware interlocks that prevent hazardous system actions and automation of user tasks.

D.6.1.2 Error Reduction

Four approaches apply to error reduction:

- Reduce Slips and Rule-based Mistakes: Rather than forcing the user to choose safe actions, design the user procedures, interface and training so that the chance of the user making an unsafe choice is low. Such reduction measures are sometimes referred to as *forcing functions*. Such a design approach is weaker than forcing safe actions.

- Clear Displays: Provide the user with a clear representation of the system's modes and state. The display of information to the user provides the user's view of the state of the system. The user's ability to diagnose and correct failures and hazards depends on the clarity and correctness of the information that is displayed.

- User Awareness: The user should be involved in the routine operation of the system. This ensures that the user's mental model is current and de-skilling does not occur [10]. When a failure occurs,

the user will be better able to respond to the failure in a constructive and correct manner that restores the system to a safe state.

- **Knowledge-based Reasoning: Provide the user with memory aids and training to support knowledge-based construction of plans.** If the user does not have an existing rule for the situation with which they have been confronted, they will have to construct a rule from their broader knowledge base.

D.6.1.3 Error Recovery

Three approaches apply to error recovery:

- **Safe State: Tolerate user errors by enabling the user to recognise potential hazards and return the system from a potentially hazardous state to a safe state.**

- **Warn: If the system has entered (or is about to enter) a hazardous state, notify the user of the state that the system is currently in and the level of urgency with which the user must act.**

- **Independent information/Interrogation: Allow the user to check the correctness of warnings and double-check the status of the system through multiple independent sources.** If the state of the system can only be judged from one source, the system has a single point of failure and the user may either believe there to be a hazard when there is none or believe there to be no hazard when one exists.

D.6.1.4 Design Principles for General Usability

D.6.1.4.0.1 - Task Efficiency: Software should help users of varied experience levels to minimise effort in performing their tasks. Automation can help to improve efficiency.

D.6.1.4.0.2 - Reuse: Ensure that users can reuse existing effort and knowledge. Knowledge reuse is usually achieved via consistent look-and-feel.

D.6.1.4.0.3 - User-Computer Communication: Facilitate collaboration between humans and computers by appropriately representing changes to the system (these changes may have been caused by either humans or computers).

Design for Human-Computer Interaction—January 2000

Michael Mahemoff and Lorraine Johnston

D.7.1 Alarm Application — Background

The application is a user-interface for an alarm system within a factory. The system allows a user to monitor the status of sensors.

D.7.2 The Hardware

Sensors are placed at various points in the factory. There are four kinds of sensors:

- Smoke sensors (to indicate fires)
- Water sensors (to indicate flooding)
- Heat sensors
- Humidity sensors

The sensors are connected to the user-interface computer.

There is an alarm associated with each sensor, e.g. a heat alarm will activate if *any one* of the heat sensors reaches a dangerous value. The four alarms are not controlled by the user-interface, but you may want to consider their presence in designing a solution.

D.7.3 Guidelines for using the Hardware

You should be aware of some usage guidelines:

- The operator will generally be observing the user-interface, and might also perform other tasks at the same time.
- When a sensor approaches, or exceeds, a critical value, the operator will probably need to go and deal with the situation.

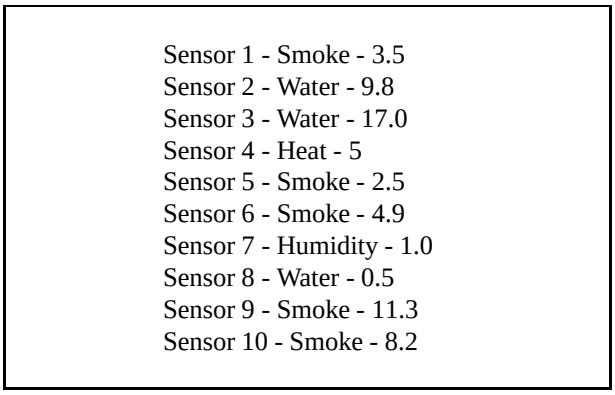


Figure D.1. Safety-critical example case study: Initial design sketch of alarm system

D.7.4 Initial Design Sketch

Figure D.1 shows a rough prototype sketch created by a designer. In this prototype, the states of each sensor is simply shown via a text interface. The user can peform several tasks via the command line:

DIAG Auto-diagnoses the system and returns the result.

LOG Outputs a log of recent system events.

STATUS *sensor number* Shows status of sensor indicated by *sensor number* (i.e. is it working?)

TEST *sensor number* Self-tests sensor indicated by *sensor number* and outputs the result.

D.8.1 Sample solution for Alarm case study

Note: This information was used by the administrator to present a sample solution. It was not provided as a handout to the subjects.

The solution presents a plan view of the factory. Each detector type is represented by a different shape. The patterns shown below indicate further design decisions.

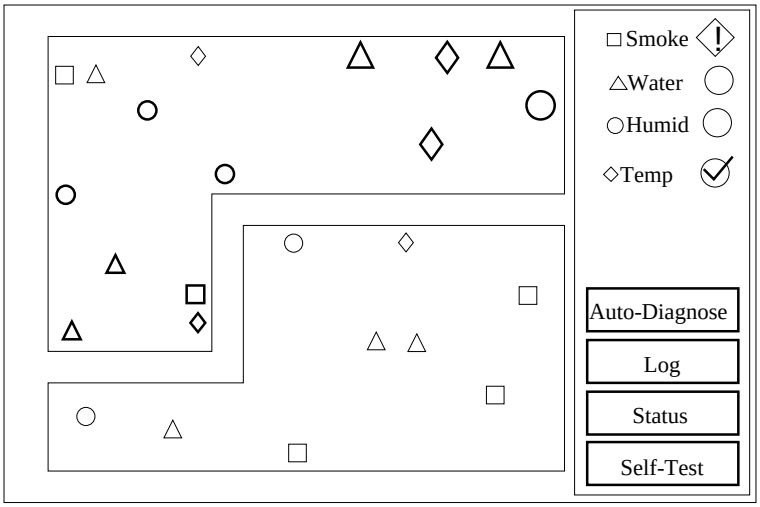


Figure D.2. Safety-critical example case study: Sample solution for Alarm problem

Task Conjunction: The user acknowledges they have dealt with a dangerous situation by "Alt-Clicking" the group checkbox. This constitutes a conjunction because both the keyboard and the mouse are used simultaneously; thus, it is unlikely to occur by accident. The reason for this joint mechanism is that it could be dangerous for the user to incorrectly indicate a hazard has been dealt with.

Affordance: The display uses a touch screen with rendering of ward layout and selection of alarms by direct manipulation.

Separation: The sensors and group checkboxes are sufficiently separate from each other to ensure the user would not accidentally click on one item while intending to click on another.

Warning: Alarms are audible and a distinct tone sounds for type of problem (4 different tones). Practices external to the user-interface can be used to maintain operator vigilance, such as requiring each operator to physically check the ward at regular intervals.

Reality Mapping: The system provides a close mapping to the physical reality, with the layout and positioning of sensors corresponding to the physical arrangement of patients.

Abstract Mapping: The system indicates the time since the last problem arose.

Recent Trend: The sensor status shows, among other things, a graph of value across time.

Interrogation: When the user selects a sensor by clicking on it, a small pocket of the screen shows a data block with some brief status information, such as ID number and a precise sensor value. The log function is also an example of interrogation.

Memory Aid: The cautionary icon (“!”) serves as a memory aid because it alerts the user that they still need to deal with a critical value.

Design for Human-Computer Interaction—January 2000

Michael Mahemoff and Lorraine Johnston

D.9.1 Radiology Application — Background

The application is a user-interface for a radiology application concerning laser emission to points in the brain. The users are radiologists who are competent computer users. We will simplify some of the concepts to make the design process more straightforward.

A patient needing treatment has their head fixed in a stationary position. Then two lasers, coming from various angles, each emit a beam. The radiologist aims to make the lasers meet at the area in need of treatment. At the intersection area, the energy is highest because it is the total of the energy of each beam.

D.9.2 The Hardware

The hardware consists of two lasers on a circular track. Each laser can move 360 degrees around the track. You do not need to concern yourself with the 3-dimensional aspect; you can assume that the radiologist has chosen a 2D slice at a fixed height along the patient's head.

Each laser beam has four parameters:

θ (0.0-360.0°), the angle around the track.

γ (0.0-360.0°), the angle indicating the direction of the laser.

w, (0.0-100.0 μm), the beam width.

t, (0.0-600.0s), the time the beam is emitting.

The user-interface computer will be connected to an optical sensor which indicates if the patient's head has moved significantly.

The user-interface computer will also be connected to a device capable of sampling the patient's heart-rate at five-second intervals.

D.9.3 Guidelines for using the Hardware

As the software will assist users in achieving their goal of treating the patient, you should be aware of some usage guidelines:

- The system is obviously safety-critical, with incorrect usage leading to critical injury.
- Although the patient's head is fixed mechanically, there is nevertheless a slight chance it can move, and this could lead to the lasers emitting to the wrong place. This would be a very dangerous occurrence.
- The radiologist must visually monitor the patient and also monitor their heart-rate. There may be circumstances when it would be inappropriate to continue the treatment.

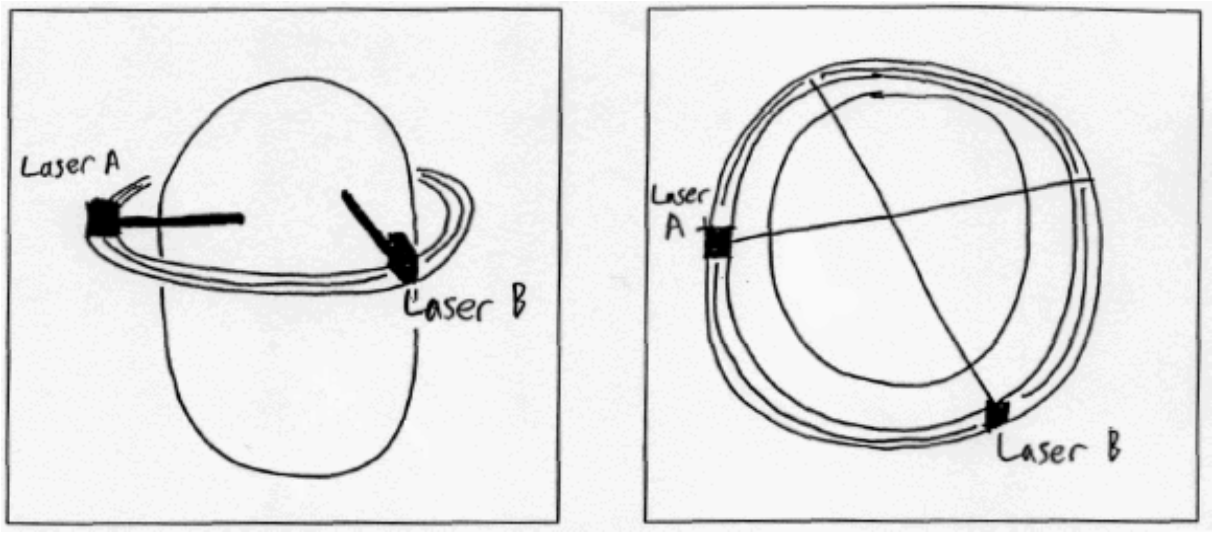


Figure D.3. Safety experiment case study: Hardware setup, showing lasers on a circular track which surrounds the patient's head. (a) perspective view, showing circular track around a slice at a fixed height. (b) top view.

- Radiologists aim for beam paths which are short in distance. This is because the beam can affect every point along the path, even though the maximum effect is when the two beams meet. (Radiologists aim for low path lengths, but do not always choose the absolute minimum path, because of factors such as skull thickness and sensitive brain regions which must be shielded.)
- The beams meet at an intersection *area*, not an intersection *point*. The beam widths can be adjusted to determine the area of intersection.
- The two lasers should start emitting at the same time and finish at the same time.

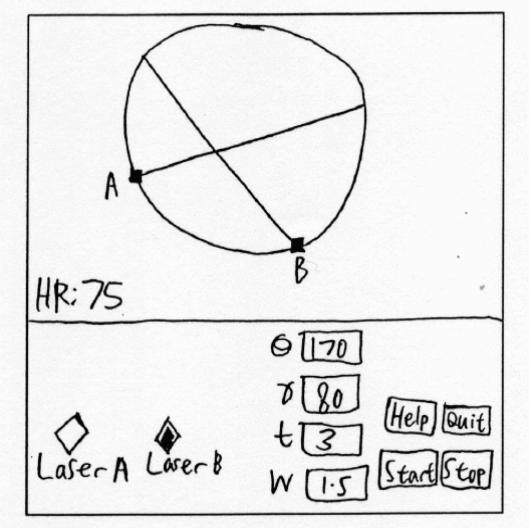


Figure D.4. Safety experiment case study: Initial design sketch of laser user-interface

D.9.4 Initial Design Sketch

Figure D.4 shows a rough prototype sketch created by a designer. In addition to the visual output, the system beeps whenever the optical sensor indicates the patient's head has moved significantly.

There is an image of a brain (this is a fictitious image; there is no hardware to show the real brain). The circular track and the approximate positions of the two lasers are shown.

The heart-rate is shown, and changes whenever a new sample is entered, i.e. every five seconds.

The control panel is below the display. A pair of radiobuttons enables the user to choose which laser's parameters are shown. The three parameters are entered via standard textboxes.

Four buttons are present on the right-hand side:

- The `Help` button leads to hypertext help.
- The `Quit` button exits the system immediately.
- The `Start` button begins beaming for both lasers immediately, according to the parameters which are set in the control panel.
- The `Stop` button stops the beaming immediately, if it is in progress.

As Figure D.5 shows, the system indicates beaming is in progress via a pop-up dialog box, which shows time elapsed and time remaining.

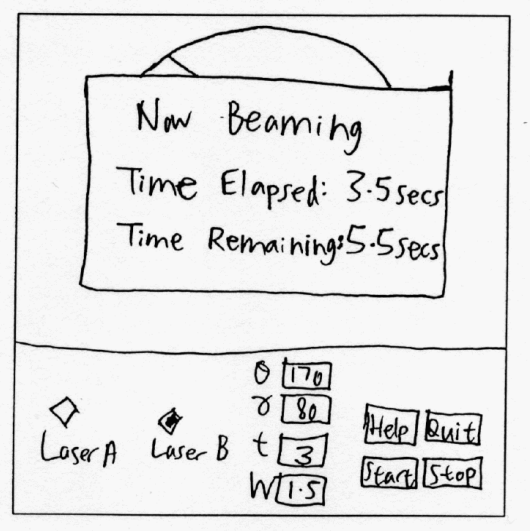


Figure D.5. Safety experiment case study: Initial design sketch of laser user-interface

Appendix E

Summarised Design Issues for Safety-Usability Study

E.1 Introduction

This appendix provides summary tables for several factors which were considered during analysis of the study on safety-usability patterns described in Chapter 7. The results are discussed in 7.5.2.1. References to patterns within the tables indicate that the subject identified their design idea as a pattern in the safety-usability language (obviously, this applies only to patterns groups).

E.2 Summary Tables for Key Design Issues

These tables correspond to the discussion in Section 7.5.2.1.

Table E.1. Results from Safety-Usability Patterns study: Laser presentation

Team	Observation
Princ-A	Overhead view of beams passing through head. Also contains side view of head and lasers passing through. Noted that thickness could be represented (but did not draw it).
Princ-B	Overhead view of beams passing through head, and also angle markings around the head.
Princ-C	Overhead view of beams passing through head, with thickness represented. Physical laser beams are coloured differently, and user interface colours reflect this.
Princ-D	Overhead view of beams passing through head, with area of intersection point representing physical intersection area.
Patt-A	Overhead view of beams passing through head, with labels at starting point of beams identifying the laser (`Reality Mapping`). Angle markings indicate orientation around head (`Redundant Information`). Indication of beam length through head.
Patt-B	Overhead view of beams passing through head (`Reality Mapping`). User can zoom in on particular regions (`Interrogation`).
Patt-C	Overhead view of beams passing through head with shape of intersection indicated and colour distinguishing between lasers (`Reality Mapping`). Also a perspective view which indicates beam directions and intersection point.
Patt-D	Overhead view of beams passing through head, with area of intersection represented (`Reality Mapping`). On both parameters and head display, Laser A is labelled with a circle and Laser B as a Triangle. Beam width reflected on display (`Reality Mapping`). Can obtain cross-sectional area via a button.

Table E.2. Results from Safety-Usability Patterns study: Head presentation and monitoring

Team	Observation
Princ-A	Critical regions of brain shown. Overhead view of head shows nose position. Side view of head shows position of 2D slice. Automatic interruption when excessive head movement detected, possibly disabling start button when this occurs.
Princ-B	Critical regions of brain shown. Overhead view of head shows nose position. Head movement level indicated by meter which can automatically stop if excessive.
Princ-C	Circle representing head. Verbally suggested that user could mark critical regions (did not draw on solution). Warns if head movement level exceeds threshold.
Princ-D	Circle representing head, and capable of zooming in and out. Head movement shown as an arrow with length proportional to physical movement. User can enter critical head movement distance, beyond which firing is interrupted.
Patt-A	Brain-like shape representing head, with critical regions shown (`Reality Mapping`). Shape of brain suggests orientation. Firing stops if optical sensor indicates that head movement level has exceeded threshold (`Interlock`).
Patt-B	Firing stops if head movement is excessive (`Interlock`). Image shows if beams will cross critical regions. Will also receive a warning if crossing occurs (`Warning`, `Redundant Information`).
Patt-C	Circle representing head. Also shows Perspective View indicating orientation. View of head moves to indicate any head movement, and interrupts firing if movement exceeds threshold (`Shutdown`). Warns user if beam will pass through critical region (`Automation`, `Warning`).
Patt-D	Brain-like shape representing head, with critical regions shown (`Reality Mapping`). As head movement reaches critical value, auditory warning begins and increases in intensity. In addition to visual display, distance from head is shown as a figure (`Abstract Mapping`). When warning begins, dialog box checks that user is willing to continue (`Task Conjunction`). Then interruption occurs beyond critical value.

Table E.3. Results from Safety-Usability Patterns study: Heart-rate monitoring

Team	Observation
Princ-A	Graph of heart-rate across time, with line showing warning region. User can set threshold for this warning region. Any values above line cause warning.
Princ-B	Heart-rate shown numerically and as a graph across time. Interrupts firing if excessive.
Princ-C	Heart-rate shown as a number below a heart icon. Icon flashes if critical value exceeded.
Princ-D	Heart-rate shown as meter with warning area and critical area, along with graph across time.
Patt-A	Graph of heart-rate across time (`Trend`), as well as instantaneous rate shown. User can set upper and lower boundaries, beyond which automatic interruption occurs (`Interlock`).
Patt-B	Heart-rate shown on dialog box which pops up during firing. Graph across time also shown on this box (`Trend`). Automatic interruption for very abnormal rates, warning for lower rates.
Patt-C	Heart-rate shown as instantaneous figure. Provides auditory warning if heart-rate drops (`Warning`), which sounds different to head movement warning (`Distinct Interaction`). XXX technically incorrect interp of DistinctInteration
Patt-D	Graph of heart-rate across time, as well as instantaneous rate shown (`Trend`).

Table E.4. Results from Safety-Usability Patterns study: Mechanisms for starting, stopping, and quitting

Team	Observation
Princ-A	Provides warning if user tries to direct beam through critical region. Colour scheme of entire view changes to show when firing is underway. Progress bar shows time elapsed and time remaining. Beaming can be stopped by hitting any key and also by clicking on a large on-screen stop button. Quit only accessible via hidden key combination to avoid accidentally quitting and ensure only administrators can quit. Quit disabled while firing.
Princ-B	Simple dialog box prompts user to confirm "Start" action. A progress bar shows time elapsed and time remaining. Graph of heart rate and head movement level also shown. Large stop button can be used to stop.
Princ-C	Start and stop with same button. Provides progress bar while firing. Provides status information indicating duration of most recent beam activity and reason for beam finishing.
Princ-D	Large Start button and large Stop button on Ready Screen. Afer clicking "Next" from parameter entry page, system will check accuracy of information of this page.
Patt-A	Start using Start button, and a popup dialog checks for confirmation (`Task Conjunction`). During firing, shows progress bar, progress percentage, time elapsed, and total time. User can cease firing via large on-screen button, as well as a physical Stop button (`Recover`). A text area shows status of last beam session (e.g. OK, head moved, laser parameters incorrect). Quit button physically separated from Help button (`Separation`).
Patt-B	Upon pushing "Accept" Button, user is presented with confirmation box showing all parameters. User can then choose "Accept" or "Back" (`Stepladder`). While firing, popup box shows time elapsed, time remaining, beam intensity, and heart-rate. Heart-rate across time (`Trend`) can be shown upon request (`Interrogation`). User can cease firing via large Stop button on this box.
Patt-C	User enters parameters then views a simulation, which is overlaid on the image of the patient's head. In the simulation, the beams are animated and the intersection point highlighted. The Start button is disabled until the simulation is being shown (`Behaviour Constraint`). When the user presses the Start button, a warning is shown (`Task Conjunction`). The lasers begins firing once the user presses OK. The user can push Stop button to cease firing.
Patt-D	User initiates firing via Start button. System confirms with user (`Task Conjunction`). Status of lasers (e.g. "Beaming") is always present. While firing, shows progress bar, progress percentage, time elapsed, and total time (`Memory Aid`). Can also view log of events via Log button. Beams are presented in an alternative colour whenever they are firing. When user chooses to quit, provided with confirmation dialog box (`Task Conjunction`).

Table E.5. Results from Safety-Usability Patterns study: Parameter Entry

Team	Observations
Princ-A	Beam time consolidated into one field covering both lasers. Suggested separate entry fields before and after decimal points (did not draw this). Descriptive parameter names and units of measure.
Princ-B	Will also warn if about to operate through critical area. Colour of beams change when user tries to enter critical region.
Princ-C	Beam time consolidated into one field covering both lasers. Descriptive parameter names and units of measure.
Princ-D	Descriptive parameter names and units of measure.
Patt-A	Beam time consolidated into one field covering both lasers, and arrow widgets available to increment/decrement each time unit, i.e. arrow pairs for seconds, minutes, hours (`Affordance`, modelled after Dix's syringe example [71]). Angles can be entered textually or by dragging lines (`Affordance`). Presentation of paramters not articulated. Checks sanity of laser angles (`Behaviour Constraint`/`Automation`). For instance, whether or not the intersect inside the brain region.
Patt-B	Beam time consolidated into one field covering both lasers. User manipulates angles by choosing angles on a circular control, entering number into a text field, and/or clicking on increment/decrement buttons. User specifies beam width and intensity by clicking on a linear bar, entering number into a text field, and/or clicking on increment/decrement buttons. Beam width bar has tapered appearance, whereas Intensity Bar has rectangular appearance (`Distinct Interaction`). Undo button allows operations to be reversed (`Recover`).
Patt-C	Checks parameters when user attempts to perform simulation. If hazardous, warn user and send back to edit mode (`Recover`).
Patt-D	Beam time consolidated into one field covering both lasers. Constraints are automatically checked before system starts beaming. These constraints can also be checked manually via a provided Check button (`Interrogation`). The constraints check includes a check to ensure lasers intersect inside brain, and a check to ensure lasers point into brain.

Appendix F

Materials for Generic Task Brainstorming Experiment

F.1.1 Introduction

The following pages contain the materials that were provided to subjects during the safety-usability study described in Section 7.5. The materials are:

Study Overview Study overview for ethics purposes.

Consent Form Consent form for ethics purposes.

Instructions Instructions to subjects (Subjects only received the information about generic tasks in the second phase of the study).

Example System Example specification and solution.

Shopcart Low-fidelity screenshot sketches for Shopcart application.

Ready Reader Low-fidelity screenshot sketches for Ready Reader application.

Section numbering within the materials has been updated for publication here, to reflect location within the appendix.

F.2.1 Information Sheet provided to subjects prior to session

The University Of Melbourne

Computer Science and Software Engineering Department

Task Design for Human-Computer Interaction: Information Sheet

Investigators: Michael Mahemoff (moke@cs.mu.oz.au), Lorraine Johnston (ljj@cs.mu.oz.au).
Telephone: 9344-9100

This sheet is an overview of the study we are conducting.

Software usability is becoming an important issue for developers. Developers would like techniques which enable them to create software which meets users' needs, and lets users interact with software easily and efficiently. Thus, various researchers have proposed techniques which developers can use to improve usability.

At present, we are considering two possible approaches to designing for usability. "Design" in this context means considering how the user interacts with the system (as opposed to software design, e.g. object-oriented or Jackson Structured design). We would like to determine how useful the design approaches are, and see what kinds of designs people produce. Therefore, we will introduce you to each approach and ask you to apply it to a couple of small re-design problems. There are no right or wrong answers. We are evaluating how effectively the approaches can be used by a designer, rather than evaluating the designers themselves. At the end, the administrator will clarify your designs and ask you about how you found each technique.

It is anticipated that the entire session will take 1-1$\frac{1}{2}$ hours. As required by the University's ethics guidelines, you are free to leave at any time and you may also withdraw your work at the same time. Not participating in this study, or choosing to leave early, will not affect your results or assessment in any way.

If you have any questions at this stage, please ask the administrator. You may also ask the administrator to clarify any information at any time during the session.

You may retain this sheet. To ensure that everyone has the same expectations going into this session, please do not show it to anyone, or discuss the study with anyone else who will be participating, for a period of four weeks. You may also wish to receive feedback on the overall study once it has been completed. To do this, either leave your email address on the consent form, or mail Michael Mahemoff (moke@cs.mu.oz.au) at any time between April 5 and December 31, 1999.

F.3.1 Consent form provided to students at start of session

The University Of Melbourne

Computer Science and Software Engineering Department

Consent form for persons participating in Task Design research project

Project: Task Design for Human-Computer Interaction.

Investigators: Michael Mahemoff (moke@cs.mu.oz.au), Lorraine Johnston (ljj@cs.mu.oz.au).

Name of participant: _____

1. I consent to participate in the above project.
2. I am aware that this project satisfies the University of Melbourne Research Ethics Committee guidelines for projects involving human subjects.
3. I have read the accompanying information sheet and I have had the opportunity to ask any questions about this study.
4. I authorise the investigators to use my results to assist them in the study.
5. I acknowledge that:
   (a) I am free to withdraw from the project at any time during this session, and if I do so, I may also withdraw any work I have performed during the session.
   (b) The project is for the purpose of research only, and is being used to evaluate design approaches, rather than to assess my own skills in any way.
   (c) The information I provide will be kept confidential, subject to any legal requirements, and my name will not be attached to the work I produced.
   (d) This consent form will be kept, among other consent forms, separate from the results produced by myself and others. In the extremely unlikely event that the investigators are asked to demonstrate that the results have not been fabricated, I may be asked to verify that I participated in this study.

6. I can provide my e-mail address if I wish to obtain feedback about the study once results have been analysed; otherwise, it is not required.

   E-mail Address (Optional): _____

   Alternatively, you can mail Michael Mahemoff (moke@cs.mu.oz.au) at any time between April 5 and December 31, 1999, to receive feedback.

Signature: _____ **Date:** ___ / ___ / _____

dd mm yyyy

Software Usability Study—-February 1999

Michael Mahemoff and Lorraine Johnston

F.4.1 EXPERIMENT OVERVIEW

We are interested in techniques which help people produce more usable software. To help us evaluate some of our methods, we will introduce you to some of these techniques, show you some initial design sketches, and ask you to improve on the sketches using the techniques. Please remember that we are evaluating the ease and effectiveness of our techniques, rather than assessing your own skills.

The study is in two parts—when you have finished Section 2, materials for the second part will be provided.

F.4.1.1 What is usability?

Usability is a quality associated with the interaction between users and computers. The ISO-9241 standard defines usability as follows:
Usability: The *effectiveness*, *efficiency* and *satisfaction* with which specified users achieve specified goals in particular environments [Italics added].

F.4.1.2 What aspects of software contribute to usability?

Perhaps the most obvious influence on usability is the user interface. A user interface should present concepts as intuitively as possible. It should also help the user to interact with the software quickly and smoothly.

In this study, we consider a factor which can be even more important than the user interface: the tasks which the user can perform with the software. The task of printing, for example, is essential for most users of word-processors. A user might be able to add and remove text easily, but it will not be very useful if it can't print. No matter how easy software is to use, usability will rate poorly for many people if it does not help them achieve their real-world goals. In the usability definition above, the *effectiveness* component relates to "the accuracy and completeness with which specified users can achieve specified goals...". How easily users can achieve their goals is influenced by the tasks which the software makes possible.

Thus, usability comes from the range of tasks users can perform on a system, as well as the ease and smoothness with which that task can be performed.

F.4.2 REDESIGNING THE SYSTEM FOR USABILITY

F.4.2.1 The materials we will provide

We are asking you to build on existing designs for two hypothetical systems. For each system, we will tell you its purpose and provide you with some initial user-interface sketches along with the tasks that are currently possible. There is one sketch for each major view in the system.

F.4.2.2 The outputs we would like you to produce

We would like you to look at the lists of possible tasks for each screen and think about which additional tasks might be useful, i.e. if the software and the user-interface was to be redesigned, what tasks would you add?

You will be asked to write down the additional tasks. You do not have to show how the user-interface would need to be modified to accomodate the new tasks (e.g. if you suggest that the user can copy a file, you do not have to create a “copy file” menu item). If you feel that it is essential to draw the new user-interface in order to clearly explain the task, then you may do so. Also, some tasks may necessitate additional screens. In such cases, you may draw the new screen design.

F.4.2.3 How to add new user tasks to the designs

We would like you to add new tasks to the existing designs. The following steps may help you, although you are asked to record any other tasks which come to mind. While working on each specification, you can return to previous points if necessary. Please use point form for all answers, and label the points as described in each section below. However, you should complete the first specification in entirity before moving onto the second specification.

You have been provided with an example re-design which will illustrate each of these steps.

Expected time: 10-20 minutes per specification (no strict time limit; finish working when you are satisfied nothing new can be added).

1. Consider how the user might interact with the system. Why would they be using the system? You can answer this question based on the general system description, rather than the current design. The current design might not yet allow the user to do everything they want with it.

   The user profile, contained at the top of the design task description, may help here. Also, note that there is often a very obvious reason why someone would use a system, but there may also be other reasons why people occasionally use the system.

▶ Identify different goals, i.e. reasons why people might be using the system (in point form as G1, G2, etc.).

2. For each goal, what does the user need to do? Imagine different paths through the system which enable the user to achieve their goal. Identify any problems the user may have, based on these reasons to use the system, or any other ideas. This may be because it is impossible or slow to do certain things. For example, the user may need to spend a long time scrolling through a document to find a word, when a search function would eliminate that problem.

   ▶ Identify any problems the user may have with the current design (in point form as P1, P2, etc.) If a goal identified in the previous section inspired you to think of the problem, identify the goal (e.g. "G2").

3. Identify new tasks which should be possible with the system. The goals and problems you found previously might suggest new tasks, and you are free to add any other ideas you have. It might also help to think about other software you have used when thinking about extra tasks.

   If you are uncertain whether the current user interface already supports a task, write the task down anyway.

   ▶ Record any new user tasks you can think of which might improve usability (in point form, as T1, T2, etc.). Give the name and a brief description of the task. If a problem or goal from the previous steps inspired you to think of this task, identify it (e.g. "P2", "G2"). There is no need to show the new user-interface if the new task could be accommodated in a fairly standard way, e.g. by adding a button, a menu entry, or a text entry field.

   Note: In most cases, it will not be necessary to draw a new user-interface. Only draw a new screen design if your task(s) imply that: (a) a new screen should be added, or (b) a radical change to a current screen is required.

F.4.3 ANOTHER APPROACH: GENERIC TASKS

We would now like you to build on your work using another approach: generic tasks. These are tasks which recur across different software systems. For example, the task of "Search" is used in word-processors, spreadsheets, internet search engines, and so on. Looking at these generic tasks may lead you to think about new tasks which may be useful in your specifications. The list is supposed to be a brainstorming tool, rather than a rigid technique. Although the generic task, "Search", would usually lead to tasks relating to searching, it is also possible that it might spark ideas not directly related to searching, such as moving through the document (because searching sometimes means moving through the document). In the process below, it is fine to document tasks which are not directly related to the generic task you are considering.

As with the previous design work, apply the following process to the first specification until it is completed, then repeat it for the second specification. Again, there is an example provided.

When you are finished, we will conclude with a brief discussion session to find out your reaction to this study.

Expected time: 5-10 minutes familiarising yourself with the generic task list, then 10-20 minutes per specification (no strict time limit; finish working when you are satisfied nothing new can be added).

1. Spend a few minutes familiarising yourself with the generic task list (of course, there is no need to do this for the second specification).

2. Step through the task list sequentially and fairly quickly. For each task, consider how it might relate to the system you have previously re-designed. Does the generic task give you any ideas for the system that were not considered in the previous re-design?

   You should not expect to find a new task for every generic task in the list. By the same token, some generic tasks may lead to more than one design change. For example, the existing design might contain a "Copy" task in one screen design, but it may be possible to copy objects of a different type in a different part of the program. Even if part of the design reflects the generic task, it may still be useful in other screens as well.

   ▶ Look at each generic task in sequence. For each generic task, record the new task(s), if any, which are appropriate for this system, as well as a brief description (like you did in the previous specification). Also, note the generic task which led you to consider this task, even if it is not directly related. As before, you can redraw the user-interface if a new screen is required or if the existing screen designs are radically altered. **Note:** When you have finished considering a task—whether or not it led to a change in your design—tick it off for reference.

   ▶ When you have ticked off all tasks in the list, flick through the list again and add any other tasks you can think of, whether they relate directly to generic tasks or not.

F.4.4 FINISHING UP

The administrator will clarify your work and enquire what you thought of the different approaches. Thankyou for spending some time to help us with this study.

F.5.1 Sketches for example system

Software Usability Study—Initial Design Sketches

0 Address Book *SAMPLE ONLY*

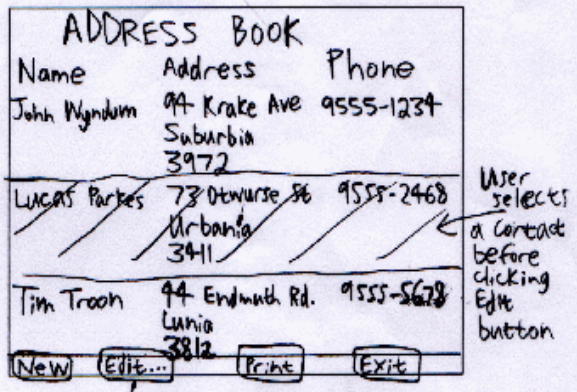
Description: Address Book lets the user record basic details—name, address, and phone number—of people they deal with.

User Profile: The software will be used by workers, who will access it from the PC on their workplace desk.

Design Constraints: Address Book will not be networked; there is no need to consider sharing of data between users. Also, only the current data fields will be used; fields such as fax and email are unnecessary. Instead, the design should provide a way for users to more easily work with the present data fields.

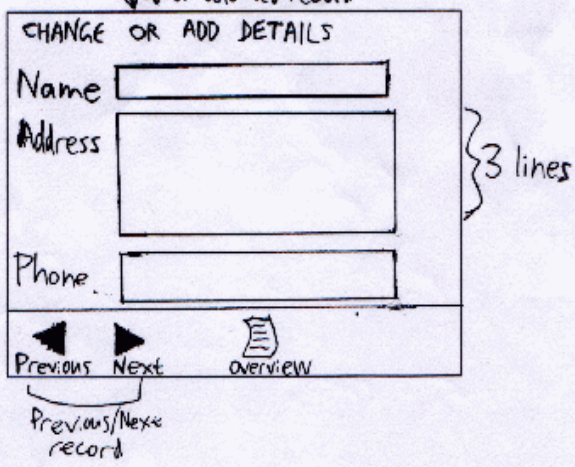
Screen S1: Overview.
CURRENT TASKS:

- Create a new entry (produces S2 with blank fields).
- Edit an existing entry (produces S2 with appropriate fields).
- Print a formatted report.
- Exit the program.



Screen S2: Details Entry.
CURRENT TASKS:

- Enter details (name, address, or phone number).
- Move to previous or next record.
- Return to overview.



F.6.1 Shopcart sketches

1 ShopCart

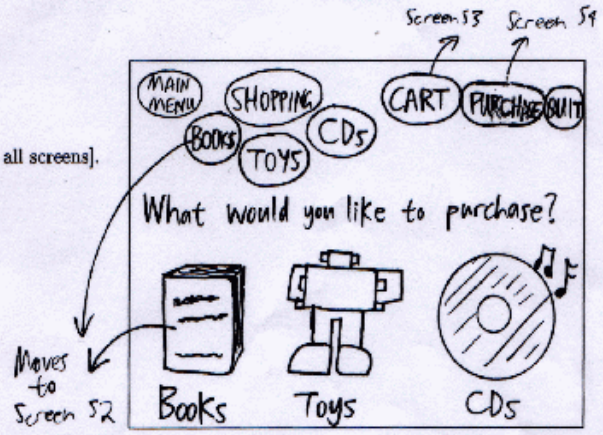
Description: Shopcart lets the user browse through a website, placing items into a shopping cart, and pay by credit card. it is a stand-alone program which connects to the internet, i.e. the user does not access it through a web browser.

User Profile: Shopcart is targeted towards internet users who want to order via the net, primarily to save time shopping at physical locations. Most users will connect via standard PCs.

Design Constraints: Whatever changes apply to the "Books" screen should apply to "Toys" and "CDs" screens; therefore, do not consider "Toys" and "CDs" screens.

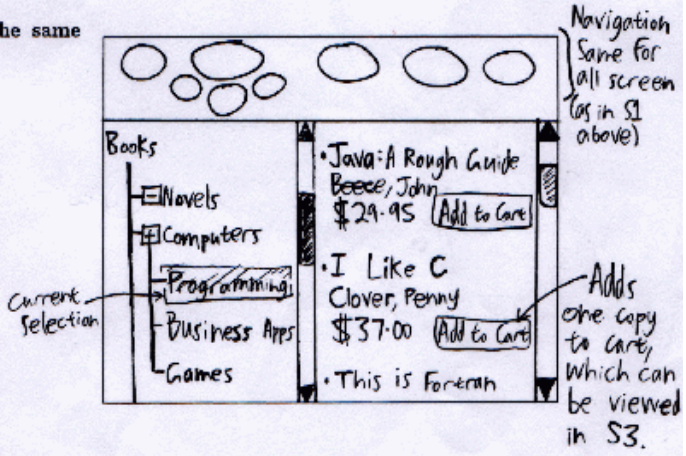
Screen S1: Main Menu.
CURRENT TASKS:

- Exit program [Applies to all screens].
- Navigate; move to other screens [Applies to all screens].



Screen S2: Books (Toys and CDs are in the same format).
CURRENT TASKS:

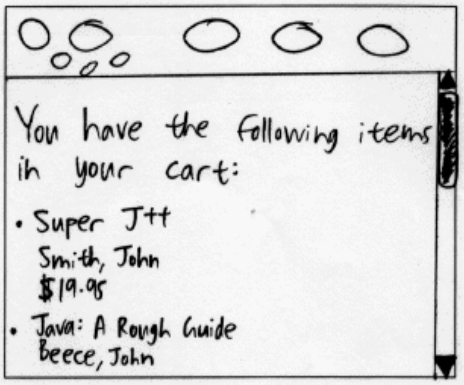
- Navigate [Same as S1].
- Change selected book category (by clicking on tree controls).
- Add book to cart.



1 ShopCart (Continued ...)

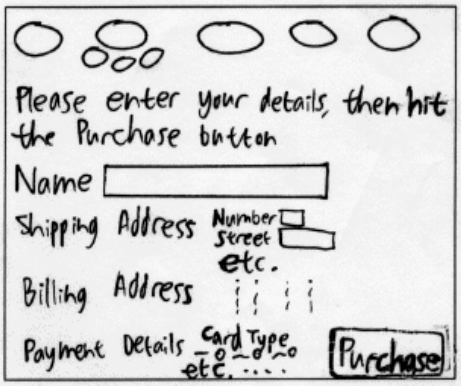
Screen S3: Cart Contents.
CURRENT TASKS:

- Navigation [Same as S1].



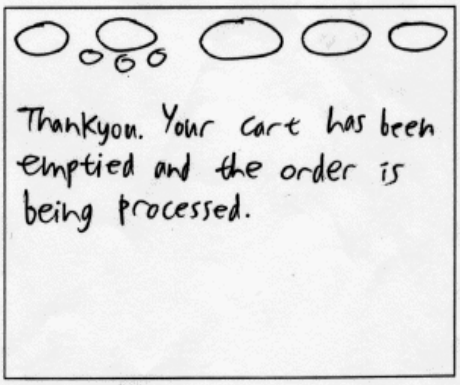
Screen S4: Purchase Screen.
CURRENT TASKS:

- Navigation [Same as S1].
- Enter details.
- Confirm details (OK button)



Screen S5: Thankyou Screen.
CURRENT TASKS:

- Navigation [Same as S1].



F.7.1 Ready Reader sketches

2 Ready Reader

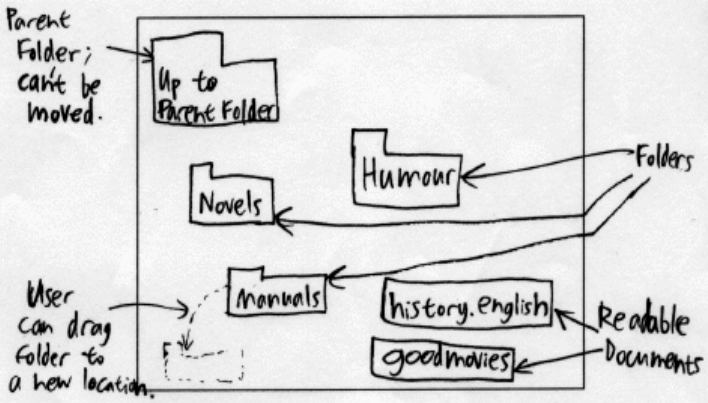
Description: Ready Reader is a palm-sized device which users can carry around with them. Users copy files, such as the text of a novel, from their PC and can then read them wherever they go. They interact using a mouse-like device and the software enables a small keyboard to be displayed, so some typing is possible. Within the reader, the files are stored in a hierarchical directory structure. As with Apple and MS-Windows environments, directories are referred to as "folders", and can contain files or other directories.

User Profile: Ready Reader is designed for anyone who spends significant time reading. It would be especially helpful for professionals and students who use public transport frequently, and would also enable people to read away from their PCs, e.g. sitting in the garden or lying in bed.

Design Constraints: (1) You can assume the device will always retain its memory; there is no need to save anything to disk; (2) You do not need to consider how the files are copied from PC; just assume they are already stored on the Reader. (3) it is not intended that users edit documents with Ready-Reader; it should be designed to help users access and organise the information in the files, rather than change that information.

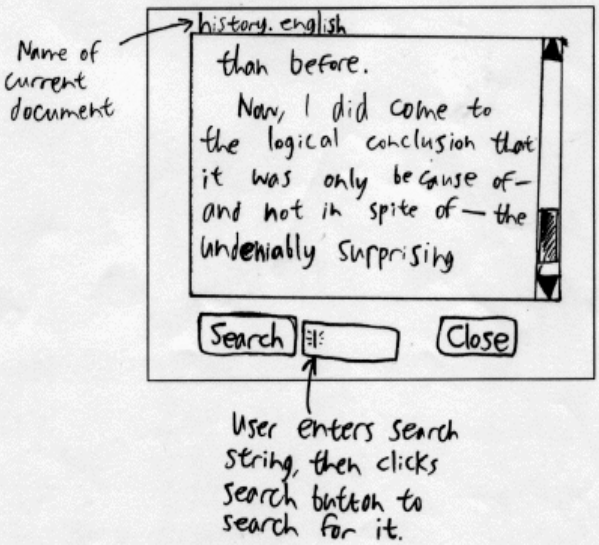
Screen S1: Folders.
CURRENT TASKS:

- Open folder (by double-clicking on it).
- Move folder (drag it into another folder or change its co-ordinates within the current folder).
- Move up to parent directory.



Screen S2: Browser.
CURRENT TASKS:

- Scroll through document.
- Close document.
- Search for text.



Appendix G

Sample Output from MMVCCreator

This appendix shows the source code generated by the MMVCCreator tool documented in Section 10.3.2. The tool takes a specification of a model and its views, and produces a code base which the developer can tailor and extend to their needs. The sample output is included here to illustrate the kind of code which the tool is capable of generating.

The inputs which were used to create the source code here are shown in Figure 10.8. They correspond to the creation of OrderModel.java, OrderViewBasic.java, and OrderViewDetailed.java, using the same properties as in the Prescribe application. Only OrderModel.java and OrderViewBasic.java are shown here; OrderViewDetailed.java is similar to OrderViewBasic.java, and was therefore ommitted for the sake of brevity.

G.1 OrderModel.java

```
/* V1.0 */
/* Class File initially generated by ModelCreator */

package med;

// EDIT THIS
import java.beans.*;
import java.util.*;

public class OrderModel implements Model {
```

```
    // Javabean Helper Class
    private PropertyChangeSupport thePropertyChangeSupport =
        new PropertyChangeSupport(this);

    // Javabean Properties
    private DrugModel drugModel;
    private Date start;
    private Date stop;
    private Integer freq;
    private Double dosage;
    private VectorModel adminVectorModel;

/* CONSTRUCTORS */

    public OrderModel(DrugModel _drugModel, Date _start, Date _stop, Integer _freq,
    Double _dosage, VectorModel _adminVectorModel ) {
        setDrugModel(_drugModel);
        setStart(_start);
        setStop(_stop);
        setFreq(_freq);
        setDosage(_dosage);
        setAdminVectorModel(_adminVectorModel);
    }

/* PUBLIC METHODS */

    public OrderModel getThis() { return this; }
    public void setThis(OrderModel _orderModel) {
        setDrugModel(_orderModel.getDrugModel());
        setStart(_orderModel.getStart());
        setStop(_orderModel.getStop());
        setFreq(_orderModel.getFreq());
        setDosage(_orderModel.getDosage());
        setAdminVectorModel(_orderModel.getAdminVectorModel());
    }
```

```
public DrugModel getDrugModel() { return drugModel; }
public void setDrugModel(DrugModel _drugModel) {
    firePropertyChange("drugModel", drugModel, _drugModel);
    drugModel=_drugModel;
    System.err.println("DrugModel changed to "+drugModel);
}

public Date getStart() { return start; }
public void setStart(Date _start) {
    firePropertyChange("start", start, _start);
    start=_start;
    System.err.println("Start changed to "+start);
}

public Date getStop() { return stop; }
public void setStop(Date _stop) {
    firePropertyChange("stop", stop, _stop);
    stop=_stop;
    System.err.println("Stop changed to "+stop);
}

public Integer getFreq() { return freq; }
public void setFreq(Integer _freq) {
    firePropertyChange("freq", freq, _freq);
    freq=_freq;
    System.err.println("Freq changed to "+freq);
}

public Double getDosage() { return dosage; }
public void setDosage(Double _dosage) {
    firePropertyChange("dosage", dosage, _dosage);
    dosage=_dosage;
    System.err.println("Dosage changed to "+dosage);
}

public VectorModel getAdminVectorModel() { return adminVectorModel; }
```

```
    public void setAdminVectorModel(VectorModel _adminVectorModel) {
        firePropertyChange("adminVectorModel", adminVectorModel, _adminVectorModel);
        adminVectorModel=_adminVectorModel;
        System.err.println("AdminVectorModel changed to "+adminVectorModel);
    }

    public void refresh() {
        firePropertyChange("this", null, this);
    }

    public String toString() {
        String ret=new String("OrderModel--> ");

        ret = ret + " ::: DrugModel = " + drugModel+ " ";
        ret = ret + " ::: Start = " + start+ " ";
        ret = ret + " ::: Stop = " + stop+ " ";
        ret = ret + " ::: Freq = " + freq+ " ";
        ret = ret + " ::: Dosage = " + dosage+ " ";
        ret = ret + " ::: AdminVectorModel = " + adminVectorModel+ " ";
        return ret;
    }

/* JAVABEAN METHODS */

    public void addPropertyChangeListener(PropertyChangeListener newListener) {
        thePropertyChangeSupport.addPropertyChangeListener(newListener);
    }

    public void addPropertyChangeListener
            (String property, PropertyChangeListener newListener) {
        thePropertyChangeSupport.addPropertyChangeListener(property, newListener);
    }

    public void removePropertyChangeListener
                (PropertyChangeListener newListener) {
        thePropertyChangeSupport.removePropertyChangeListener(newListener);
```

```
    }

    public void removePropertyChangeListener
            (String property, PropertyChangeListener newListener) {
        thePropertyChangeSupport.removePropertyChangeListener
            (property, newListener);
    }

    public void firePropertyChange(String propertyName, Object oldValue,
                                                        Object newValue) {
        System.err.println("Model changed.");
        thePropertyChangeSupport.firePropertyChange(propertyName, oldValue,
                                                                newValue);
        thePropertyChangeSupport.firePropertyChange("this",null,this);
    }

/* TESTING METHODS */

    public static void main(String args[]) {
        // EDIT THIS - Add arguments to constructor
        // OrderModel testOrderModel= new OrderModel();
        // System.out.println(testOrderModel);
    }
} //End of class
```

G.2 OrderViewBasic.java

```
/* V1.0 */
/* Class File initially generated by ViewCreator */

package med;

// EDIT THIS
import java.beans.*;
import java.awt.*;
```

```
import javax.swing.*;

public class OrderViewBasic extends JPanel implements View {

/* FINAL VARIABLES (CONSTANTS) */

/* PRIVATE ATTRIBUTES */

    private OrderModel orderModel;
    private OrderModelListener orderModelListener=new OrderModelListener();
    // TODO - ADD COMPONENTS TO REPRESENT ATTRIBUTES */
    // e.g. JLabel freqLabel = new JLabel()

/* CONSTRUCTORS */

    public OrderViewBasic(OrderModel _orderModel) {
        setModel(_orderModel);
        // TODO - CONFIGURE SUBVIEWS AND OTHER COMPONENTS, then add them
        // e.g. add(drugViewSummary);
        // e.g. freqLabel.setBackground(...);
        // e.g. add(freqLabel);
    }

    // UNCOMMENT THIS CONSTRUCTOR IF DEFAULT MODEL CONSTRUCTOR EXISTS
    // public OrderViewBasic() {
        // this(new OrderModel());
    // }

    public Model getModel() {return orderModel; }

    public void setModel(Model _orderModel) {
        OrderModel oldOrderModel=orderModel;
        if (oldOrderModel!=null) {
            orderModel.removePropertyChangeListener(
                "drugModel", orderModelListener);
            orderModel.removePropertyChangeListener(
```

```
            "start", orderModelListener);
        orderModel.removePropertyChangeListener(
            "stop", orderModelListener);
        orderModel.removePropertyChangeListener(
            "freq", orderModelListener);
        orderModel.removePropertyChangeListener(
            "dosage", orderModelListener);
        orderModel.removePropertyChangeListener(
            "adminVectorModel", orderModelListener);
    }
    orderModel= (OrderModel) _orderModel;
    updateAll();
    orderModel.addPropertyChangeListener(orderModelListener);
    firePropertyChange("OrderModel",oldOrderModel,orderModel);
}

public String toString() {
    String ret=new String("basic view of OrderModel");
     // TODO - Append anything relevant to ret
    return ret;

}

public void updateAll() {
    updateDrugModel(orderModel.getDrugModel());
    updateStart(orderModel.getStart());
    updateStop(orderModel.getStop());
    updateFreq(orderModel.getFreq());
    updateDosage(orderModel.getDosage());
    updateAdminVectorModel(orderModel.getAdminVectorModel());
     // TODO - APPEND ANY OTHER MESSAGES
}

public void updateDrugModel(DrugModel _drugModel) {
    // TODO - MAKE ANY UPDATES TO REFLECT CHANGES TO drugModel
}
```

```
public void updateStart(Date _start) {
    // TODO - MAKE ANY UPDATES TO REFLECT CHANGES TO start
}

public void updateStop(Date _stop) {
    // TODO - MAKE ANY UPDATES TO REFLECT CHANGES TO stop
}

public void updateFreq(Integer _freq) {
    // TODO - MAKE ANY UPDATES TO REFLECT CHANGES TO freq
}

public void updateDosage(Double _dosage) {
    // TODO - MAKE ANY UPDATES TO REFLECT CHANGES TO dosage
}

public void updateAdminVectorModel(VectorModel _adminVectorModel) {
    // TODO - MAKE ANY UPDATES TO REFLECT CHANGES TO adminVectorModel
}

private class OrderModelListener implements PropertyChangeListener {
    public void propertyChange(PropertyChangeEvent ev) {
        if (ev.getPropertyName().equals("drugModel"))
        updateDrugModel((DrugModel) ev.getNewValue());
        if (ev.getPropertyName().equals("start"))
        updateStart((Date) ev.getNewValue());
        if (ev.getPropertyName().equals("stop"))
        updateStop((Date) ev.getNewValue());
        if (ev.getPropertyName().equals("freq"))
        updateFreq((Integer) ev.getNewValue());
        if (ev.getPropertyName().equals("dosage"))
        updateDosage((Double) ev.getNewValue());
        if (ev.getPropertyName().equals("adminVectorModel"))
        updateAdminVectorModel((VectorModel) ev.getNewValue());
    }
```

```
    }
    public static void main(String args[]) {
        OrderModel testOrderModel = new OrderModel();
        OrderViewBasic testOrderViewBasic = new OrderViewBasic(testOrderModel);

        JFrame testFrame = new JFrame();
        testFrame.getContentPane().add(testOrderViewBasic);
        testFrame.setBounds(100,100,400,400);
        testFrame.setVisible(true);
        // TODO - CHANGE THE MODEL ETC.
    }
}
```

Appendix H

Glossary of Acronyms and Definitions

H.1 Acronyms

This section lists acronyms used throughout the thesis.

AWT Abstract Windowing Toolkit (See Definitions section below)

CSCW Computer-Supported Collaborative Work

FAQ Frequently Asked Question(s)

GUI Graphical User Interface

HCI Human-Computer Interaction

JSD Jackson System Development (See Section 2.3.3)

JSP Jackson Structured Programming (See Section 2.3.3)

MVC Model-View-Controller (See Definitions section below)

MMVC Multiple Model-View-Controller ((See Chapter 10))

MUSE Method for USability Engineering (See Section 2.3.3)

MARCO Multiple-Agent Real-time Clock Objects (see (See Chapter 9))

OS Operating System

OVID Object, View, and Interaction Design (See Section 2.3.3)

PAC Presentation-Abstraction-Control (See Section 9.3.1)

ROCI Repository of Online Cultural Information (See Chapter 6)

QWAN Quality Without A Name (See Section 3.3)

WIMP Window-Icon-Mouse-Pointer (See Definitions section below)

WYSIWYG What You See Is What You Get

SE Software Engineering

UI User Interface

UML Unified Modelling Language (See Definitions section below)

H.2 Definitions

Alexandrian form A common form of expressing patterns, in which the pattern is described as a set of key-value pairs (examples of keys are “Context”, “Problem”, etc.) (See Section 3.2.3).

Abstract Windowing Toolkit A graphical library provided in the standard Java library.

ASCII An encoding standard allowing for English-language characters, Arabic numerals, and various symbols.

Construction-driven design A mode of design in which design tools such as forms or procedures are used to drive designers from a problem to a solution. May be contrasted with evaluation-driven design (See Section 4.4.1).

Covert factor A cultural factor which is somewhat intangible, such as problem-solving behaviour (See Section 5.3.2.2).

Critique An application illustrating the Planet pattern language (See Section 11.2).

Detailed design See “Detailed software design”.

Detailed software design The process of planning the construction of a software program, typically involving specification of the system architecture, module contents, module interfaces, dynamic workings, and so on.

Evaluation-driven design A mode of design in which designers use haphazard approaches to move from problem to solution, then evaluate the solution, and repeat this process until a satisfactory design is achieved (See Section 4.4.1).

Generative design See “Construction-driven design”.

Generic task A task which recurs across numerous applications, such as “Add” or “Compare” (See Section 8).

High-level design The process of specifying of a system’s functionality, user-computer dialogue, user-interface appearance, and related constructs.

(Software) Internationalisation The process of producing software which will be used by users from numerous countries and/or cultures (See Section 5.3).

Locale The realisation of a cultural model in software.

Model-View-Controller An architectural pattern (or “architectural style”) for detailed software design of systems with a significant user-interface component. The intention is to provide a degree of separation between the domain model, the user-interface, and the mechanisms for user control (See Section 9.1).

MySQL A database for website generation, used in the Online Repository prototype (See Chapter 6).

Overt factor A cultural factor which is tangible and easily defined, such as problem-solving behaviour (See Section 5.3.2.1).

PHP A scripting language for website generation, used to implement the Online Repository prototype (See Chapter 6)

Piecemeal growth Gradual, evolutionary, growth which is claimed to be offered by pattern-led development (See Section 3.2.3.1).

Prescribe An application illustrating the MMVC detailed design framework (See Section 10.2).

Safety-critical system A system in which safety is a critical factor (i.e. a factor which must exceed a certain threshold for the system to be acceptable), such as a nuclear reactor or a life support system.

Swing A library provided in the standard Java library which extends Java’s Abstract Windowing Toolkit to provide more sophisticated user-interface classes.

Target culture In the context of the Planet pattern language Section 5, a culture which a software application is supporting.

Unicode An encoding standard enabling a large range of international characters to be represented.

Unified Modelling Language A popular notation for object-oriented design (see Fowler and Scott, 2000 [84]).

Window-Icon-Mouse-Pointer A paradigm of interactive system design in which the user typically uses the mouse to perform direct manipulation on a graphical user-interface. Currently the dominant paradigm in mainstream office and home systems.

Bibliography

[1] C. Alexander. *Notes on the Synthesis of Form*. Harvard University Press, Cambridge, MA, 1964.

[2] C. Alexander. *The Timeless Way of Building*. Oxford University Press, New York, 1979.

[3] C. Alexander. The origins of pattern theory, the future of the theory and the generation of a living world. *Communications of the ACM*, 16(5), September 1999. Text of Christopher Alexander's keynote presentation at OOPSLA '96. Article Introduction by Coplien, J. O.

[4] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel. *A Pattern Language*. Oxford University Press, New York, 1977.

[5] M. G. Alvarez, L. R. Kasday, and S. Todd. How we made the web site international and accessible: A case study. In *Proceedings of the 4th Conference of Human Factors and the Web*. AT & T, 1998. http://www.research.att.com/conf/hfweb/proceedings/proceedings/alvarez/. Accessed March 28, 1999.

[6] Andover.Net. Slashcode framework. http://slashcode.com/about.shtml, Accessed December 12, 2000.

[7] Apple Computer. *Human Interface Guidelines: The Apple Desktop Interface*. Addison-Wesley, Reading, MA, 1987.

[8] ISIS Information Architects. Interface Hall of Shame: Quicktime 4.0 player, 1999. http://www.iarchitect.com/qtime.htm. Accessed December 4, 1999.

[9] C. Baber. *Beyond the Desktop*. Academic Press, London, 1997.

[10] L. Bainbridge. Ironies of automation. In J. Rasmussen, K. Duncan, and J. Leplat, editors, *New Technology and Human Error*, chapter 24, pages 271–283. John Wiley and Sons Ltd., 1987.

[11] L. Bainbridge and S. Antonio Ruiz Quintanilla, editors. *Developing Skills with Information Technology*. John Wiley and Sons Ltd., 1989.

[12] S. Balbo and C. Lindley. Adaptation of a task analysis methodology to the design of a decision support system. In S. Howard, J. Hammond, and G. Lindgaard, editors, *Human-Computer Interaction: Interact '97*, pages 355–361. Chapman & Hall, London, 1997.

[13] W. Barber and A. Badre. Culturability: The merging of culture and usability, 1998. http:-//www.research.att.com/conf/hfweb/proceedings/proceedings/barber/. Accessed March 28, 1999.

[14] E. Bayle, R. Bellamy, G. Casaday, T. Erickson, S. Fincher, B. Grinter, B. Gross, D. Lehder, H. Marmolin, B. Moore, C. Potts, G. Skousen, and J. Thomas. Putting it all together. Towards a pattern language for interaction design: A CHI 97 workshop. *SIGCHI Bulletin*, 30(1):17–23, January 1998.

[15] K. Beck. Kent Beck's patterns (early development, user interface). Portland Patterns Repository. http://c2.com/ppr/. Accessed December 12, 2000.

[16] K. Beck, J.O. Coplien, R. Crocker, L. Dominick, G. Meszaros, F. Paulisch, and J. Vlissides. Industrial experience with design patterns. In *Proceedings of the 18th International Conference on Software Engineering*, pages 103–114. IEEE Computer Society, Los Alamitos, CA, 1996.

[17] K. Beck and W. Cunningham. Using pattern languages for object-oriented programs. Technical Report CR-87-43, Tektronix. Presented at the OOPSLA '87 Workshop on Specification and Design for Object-Oriented Programming.

[18] M. Belge. The next step in software internationalization. *Interactions*, 2(1):21–25, January 1995.

[19] S. Berczuk. Hot topics: Finding solutions through pattern languages. *IEEE Computer*, 27(12):75–76, December 1994.

[20] H. Beyer and K. Holtzblatt. *Contextual Design: Defining Customer-Centered Systems*. Morgan Kaufmann, San Francisco, 1998.

[21] F. Bodart, A. Hennebert, J. Leheurex, I. Provot, B. Sacre, and J. Vanderdonckt. Towards a systematic building of software architecture: The Trident methodological guide. In P. Palanque and R. Bastide, editors, *Design Specification, and Verification of Interactive Systems*, pages 237–253. Springer, Wien, 1995.

[22] B. W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5):33–44, July 1988.

[23] B. W. Boehm, A. Egyed, J. Kwan, D. Port, A. Shah, and R. Madachy. Using the winwin spiral model: A case study. *IEEE Computer*, 31(7):61–72, May 1988.

[24] J. Borchers. *A Pattern Approach to Interaction Design*. John Wiley & Sons, New York, 2001.

[25] J. O. Borchers. Designing interactive music systems: A pattern approach. In H. Bullinger and J. Ziegler, editors, *HCII '99: 8th International Conference on Human-Computer Interaction*, pages 276–280. Lawrence Erlbaum Associates, London, 1999.

[26] J. O. Borchers. Interaction design patterns: Twelve theses, 2000. Position Paper for the CHI 2000 Workshop on Pattern Languages for Interaction Design: Building Momentum. http://www.tk.uni-linz.ac.at/~jan/publications/index.html. Accessed June 12, 2000.

[27] M. Bottomley. A pattern language for simple embedded systems. In *Pattern Languages of Programs 1999 Proceedings*, Monticello, IL, 1999. http://jerry.cs.uiuc.edu/ ~plop/plop99/proceedings/. Accessed September 5, 1999.

[28] P. Bourges-Waldegg and S. A. R. Scrivener. Meaning, the central issue in cross-cultural HCI design. *Interacting with Computers*, 9(3):287–309, January 1998.

[29] M. Bradac and B. Fletcher. A pattern language for developing form style windows. In R. Martin, D. Riehle, and F. Buschmann, editors, *Pattern Languages of Program Design 3*, pages 347–357. Addison-Wesley Longman, Reading, MA, 1998.

[30] I. M. Breedvelt-Schouten, F. Paternó, and C. A. Severijns. Reusable structures in task models. In H. D. Harrison and J. C. Torres, editors, *Design, Specification and Verification of Interactive Systems*, pages 225–238. Springer, New York, 1997.

[31] Brighton Usability Group. The Brighton Patterns Collection. Maintained by Griffiths, R. N. at http://www.it.bton.ac.uk/cil/usability/patterns/. Accessed September 4, 2000.

[32] R. Brooks. Comparative task analysis: An alternative direction for human-computer interaction science. In J. M. Carroll, editor, *Designing Interaction: Psychology at the Human-Computer Interface*, pages 50–59. Cambridge University Press, Cambridge, UK, 1991.

[33] K. Brown. Using patterns in order management systems: A design patterns experience report. *Object Magazine*, January 1996.

[34] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *A System of Patterns*. John Wiley & Sons, Chichester, 1995.

[35] M. Campione, K. Walrath, and A. Huml. *The Java Tutorial Continued*. Addison-Wesley, Reading, MA, 1999.

[36] J. M. Carroll, editor. *Scenario-Based Design: Envisioning Work and Technology in System Development*. John Wiley & Sons, New York, 1995.

[37] J. M. Carroll, W. A. Kellogg, and M. B. Rosson. The task-artifact cycle. In J. M. Carroll, editor, *Designing Interaction: Psychology at the Human-Computer Interface*, pages 74–102. Cambridge University Press, Cambridge, UK, 1991.

[38] G. Casaday. Rationale in practice: Templates for capturing and applying design experience. In T. P. Moran and J. M. Carroll, editors, *Design Rationale: Concepts, Techniques and Use*, pages 351–372. Lawrence Erlbaum Associates, Hillsdale, NJ, 1995.

[39] G. Casaday. Notes on a pattern language for interactive usability. In *CHI 97 Electronic Publications: Late Breaking/Short Talks*. ACM, 1997. http://www.acm.org/sigchi/chi97/proceedings/short-talk/gca.htm. Accessed December 29, 2000.

[40] P. M. Chisnall. *Strategic Business Marketing*. Prentice Hall International, Hertfordshire, UK, 3rd edition, 1995.

[41] M. P. Cline. The pros and cons of adopting and applying design patterns in the real world. *Communications of the ACM*, 39(10):47–49, October 1996.

[42] A. Cockburn. The interaction of social issues and software architecture. *Communications of the ACM*, 39(10):40–46, October 1996.

[43] J. Coldewey. User interface software. In *Pattern Languages of Program Design 1998*, 1998. http://jerry.cs.uiuc.edu/plop/plop98/final_submissions/. Accessed March 30, 1999.

[44] D. Collins. *Designing Object-Oriented User Interfaces*. Benjamin/Cummings, Redwood City, CA, 1995.

[45] J. Connolly. Developing a cultural model. In E. M. del Galdo and J. Nielsen, editors, *International User Interfaces*, pages 20–40. John Wiley & Sons, New York, 1996.

[46] L. L. Constantine. Persistent usability: A multiphasic user interface architecture for supporting the full usage life cycle. In S. Howard and Y. K. Leung, editors, *Proceedings of OZCHI 94*, pages 9–14. CHISIG, Ergonomics Society of Australia, Canberra, Australia, 1994.

[47] L. L. Constantine. Usage-centered software engineering: New models, methods, and metrics. In M. Purvis, editor, *Proceedings 1996 International Conference of Software Engineering: Education Practice*, pages 2–9. IEEE Computer Society, Los Alamitos, CA, 1996.

[48] L. L. Constantine and A. D. Lockwood. *Software for Use: A Practical Guide to the Models and Methods of User-Centered Design*. Addison-Wesley, 1999.

[49] L. L. Constantine and L. Lockwood. Usage-centered design through essential modeling, 1996.

[50] A. Cooper. *About Face: The Essentials of User Interface Design*. IDG Books, Foster City, CA, 1995.

[51] A. Cooper. *The Inmates Are Running the Asylum: Why High-Tech Products Drive Us Crazy and How to Restore the Sanity*. SAMS, Indiana, 1999.

[52] J. O. Coplien. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, Reading, MA, 1992.

[53] J. O. Coplien. Software design patterns: Common questions & answers. In *Proceedings of Object Expo New York*, pages 39–42. SIGS, New York, 1994. http://www.bell-labs.com/people/cope/bibliography.html. Accessed June 1, 1998.

[54] J. O. Coplien. A generative development-process pattern language. In J. O. Coplien and D. C. Schmidt, editors, *Pattern Languages of Program Design*, pages 183–237. Addison-Wesley, Reading, MA, 1995.

[55] J. O. Coplien. Idioms and patterns as architectural literature. *IEEE Software*, 14(1):37–42, January/February 1997.

[56] J. O. Coplien. Setting the stage. In L. Rising, editor, *The Patterns Handbook*, pages 301–319. Cambridge University Press, Cambridge, UK, 1998.

[57] J. O. Coplien and R. Gabriel. A pattern definition, 2000. http://hillside.net/patterns/definition.html. Accessed September 1, 2000. (Creation date unspecified.).

[58] T. Coram and J. Lee. Experiences — a pattern language for user interface design. In *Pattern Languages of Program Design 1996 Proceedings*, 1996. http://www.maplefish.com/todd/papers/experiences/Experiences.html. Accessed August 5, 1999.

[59] Dell Computer Corporation. Dell computer website, 2000. http://www.dell.com. Accessed December 27, 2000.

[60] Microsoft Corporation. Microsoft Word 97.

[61] J. Coutaz. The construction of user interfaces and the object paradigm. In J. Bézivin, J. M. Hullot, P. Cointe, and H. Lieberman, editors, *European Conference on Object-Oriented Programming (Lecture Notes in Computer Science 276)*, pages 121–130. Springer, Berlin, 1987.

[62] J. Coutaz. Architectural design for user interfaces. In J. J. Marciniak, editor, *Encyclopaedia of Software Engineering*, pages 38–49. John Wiley & Sons, New York, 1994.

[63] J. Coutaz. Evaluation techniques: Exploring the intersection of hci and software engineering. In R. N. Taylor and J. Coutaz, editors, *ISE '94 Workshop on SE-HCI: Joint Research Issues*, pages 35–48. Springer, Berlin, 1994.

[64] Cunningham and Cunningham, Inc. (Maintainer). Wikiwikiweb at the Portland Patterns Repository. http://c2.com/cgi/wiki?WelcomeVisitors, Accessed December 12, 2000.

[65] J. Cybulski and T. Linden. Composing multimedia artifacts for reuse. In N. Harrison, B. Foote, and H. Rohnert, editors, *Pattern Languages of Program Design 4*, pages 461–488. Addison-Wesley Longman, 2000.

[66] P. Danton de Rouffignac. *How to Sell to Europe*. Pitman, New York, 1989.

[67] D. L. Day. Cultural bases of interface acceptabnce: Foundations. In M. A. Sasse, R. J. Cunningham, and R. L. Winder, editors, *People and Computers X: Proceedings of HCI '95*, pages 35–37. Springer, London, 1996.

[68] F. de Souza and N. Bevan. The use of guidelines in menu interface design: Evaluation of a draft standard. In E. Diaper, D. Gilmore, G. Cockton, and B. Shakel, editors, *Human-Computer Interaction: Interact '90*, pages 435–440. Elsevier Science Publishers, Amsterdam, 1990.

[69] E. del Galdo. Internationalization and translation: Some guidelines for the design of human-computer interfaces. In J. Nielsen, editor, *Designing User Interfaces for International Use*, pages 39–44. Elsevier, Amsterdam, 1990.

[70] T. Digre. Business object component architecture. *IEEE Software*, 15(5):60–69, September/October 1998.

[71] A. Dix, J. Finlay, G. Abowd, and R. Beale. *Human-Computer Interaction*. Prentice Hall, 1998.

[72] EDS, Electrowatt Engineering Services (U.K.), Philips Analytical X-Ray, and Philips Research Laboratories and Egonomics & HCI Unit-University College London. Final report: Benefits of integrating usability and software engineering methods. ESSI project 10290, 1995.

[73] T. Elwert and E. Schlungbaum. Modelling and generation of graphical user interfaces in the tadeus approach. In P. Palanque and R. Bastide, editors, *Design Specification, and Verification of Interactive Systems*, pages 193–208. Springer, Berlin, 1995.

[74] T. Erickson. Lingua francas for design: Sacred places and pattern language. In *Designing Interactive Systems (DIS) 2000*, pages 357–368. ACM, New York, 2000.

[75] T. Erickson. Patterns languages as languages. Position Paper for CHI 2000 Workshop on Pattern Languages for Interaction Design, 2000.

[76] T. Erickson. Towards a pattern language for interaction design. In P. Luff, J. Hindmarsh, and C. Heath, editors, *Workplace Studies: Recovering Work Practice and Informing Systems Design*. Cambridge University Press, Cambridge, 2000. In Press. Online version: http://www.pliant.org/personal/Tom_Erickson/DesignLinguaFranca.html. Accessed March 4, 2000.

[77] V. Evers and D. Day. The role of culture in interface acceptance. In S. Howard, J. Hammond, and G. Lindgaard, editors, *Human-Computer Interaction: Interact '97*, chapter 44, pages 260–267. Chapman & Hall, London, 1997.

[78] T. Fernandes. *Global Interface Design*. AP Professional, Chesnut Hill, MA, 1995.

[79] R. Fields and P. Wright. Safety and human error in activity systems: A position. CHI'98 Workshop on Designing User Interfaces for Safety Critical Systems, 1998.

[80] G. Fischer. Beyond "couch potatoes": From consumers to designers. In J. Tanaka, editor, *Asia-Pacific Computer-Human Interaction (APCHI) '98 Proceedings*, pages 2–9. IEEE Computer Society, Los Alamitos, CA, 1998.

[81] C. Floyd. Outline of a paradigm change in software engineering. *ACM SIGSOFT*, 13(2), April 1988.

[82] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley, Reading, MA, 2nd edition, 1990.

[83] M. Fowler. *Analysis Patterns: Reusable Object Models*. Addison-Wesley, Reading, MA, 1997.

[84] M. Fowler and K. Scott. *UML Distilled: A Brief Guide to the Standard Object Modelling Language*. Addison-Wesley, Reading, MA, 2nd edition, 2000.

[85] A. L. Friedman. *Computer Systems Development: History, Organization and Implementation*. John Wiley & Sons, Chichester, UK, 1989.

[86] R. P. Gabriel. *Patterns of Software*. Oxford University Press, New York, 1996.

[87] R. P. Gabriel and R. Goldman. Jini community pattern language, 1999. http://jerry.cs.uiuc.edu/~plop/plop99/proceedings/. Accessed September 5, 1999.

[88] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[89] D. Gentner and J. Nielsen. The anti-Mac interface. *Communications of the ACM*, 37(8):70–82, August 1996.

[90] J. Goguen and C. Linde. Techniques for requirements elicitation. In *IEEE International Symposium on Requirements Engineering*, pages 152–164, Los Alamitos, CA, 1993. IEEE Computer Society.

[91] J.D. Gould and C. Lewis. Designing for usability: Key principles and what designers think. In A. Janda, editor, *CHI 83 Proceedings*, pages 50–53. ACM, New York, 1983.

[92] S. Grabow. *Christopher Alexander: The Search for a New Paradigm in Architecture*. Oriel Press, Northumberland, UK, 1983.

[93] Å. Granlund and D. LaFrenière. A pattern-supported approach to the user interface design process, 1999. Created as key material for a workshop at UPA 99. http://www.gespro.com/lafrenid/patterns.html. Accessed April 12, 2000.

[94] R. Green. Human error on the flight deck. *Philosphical Transactions of the Royal Society of London, Series B*, 327(1241):503–511, 1990.

[95] D. W. Green (initially the late Perry, R. H.). *Perry's Chemical Engineer's Handbook*. McGraw-Hill, New York, 6th edition, 1995.

[96] J. Grosjean. *Life with Two Languages*. Harvard University Press, Cambridge, MA, 1982.

[97] J. Grudin. The development of interactive systems: Bridging the gaps between developers and users. *IEEE Computer*, 24(4), April 1991.

[98] J. Grudin and S. Poltrock. Software engineering and the chi and cscw communities. In R. N. Taylor and J. Coutaz, editors, *ISE '94 Workshop on SE-HCI: Joint Research Issues*, pages 93–112. Springer, Berlin, 1994.

[99] E. Gülch and H. Müller. Object-oriented software design in semiautomatic building extraction. In D. M. McKeown, J. McGlone, and O. Jamet, editors, *Integrating Photogrammetric Techniques with Scene Analysis and Machine Vision III*, pages 37–48. SPIE, Bellingham, WA, 1997.

[100] G. S. Halford, W. H. Wilson, and S. Phillips. Processing capacity defined by relational complexity: Implications for comparative, developmental, and cognitive psychology. *Behavioural and Brain Sciences*, 21(6):803–831, 1998.

[101] E. T. Hall and M. R. Hall. *Understanding Cultural Differences*. Intercultural Press, Yarmouth, Maine, 1990.

[102] J. Harris and A. Henderson. Evolution in action: HCI in a world of pliant computing, 1999. At http://www.pliant.org/Papers-Area.html. Accessed October 12, 1999.

[103] H. R. Hartson and D. Hix. Towards empirically derived methodologies and tools for human-computer interface development. *International Journal of Man-Machine Studies*, 4(31):477–489, 1989.

[104] R. J. Hathaway III (Maintainer). Object FAQ (Frequently Asked Questions for Usenet comp.object newsgroup), 1998. http://www.cyberdyne-object-sys.com/oofaq2/. Accessed November 28, 2000.

[105] L. Herman. Towards effective usability evaluation in Asia. In J. Grundy and M. Apperly, editors, *Proceedings of OZCHI 96*, pages 135–136. IEEE Computer Society, Los Alamitos, CA, 1996.

[106] Hillside.com. Patterns home page. http://www.hillside.net/patterns/patterns.html. Accessed October 25, 2000.

[107] D. H. Hix and H. R. Hartson. *Developing User Interfaces*. John Wiley & Sons, New York, 1993.

[108] N. Hoft. Developing a cultural model. In E. M. del Galdo and J. Nielsen, editors, *International User Interfaces*, pages 41–73. John Wiley & Sons, New York, 1996.

[109] T. Howard. *The Smalltalk developer's guide to VisualWorks*. SIGS, New York, 1995.

[110] J. Howell (Chief Developer). FAQ-O-Matic Framework. http://faqomatic.sourceforge.net/, Accessed December 12, 2000.

[111] A. Hussey. Patterns for safer human-computer interfaces. In M. Felici, K. Kanoun, and A. Pasquini, editors, *Computer Safety, Reliability and Security (SAFECOMP '99)*, pages 103–112. Springer, Berlin, 1999. LNCS 1698.

[112] A. Hussey and D. Carrington. Comparing the MVC and PAC architectures: a formal perspective. *IEEE Proceedings of Software Engineering*, 144(4):224–236, August 1997.

[113] A. Hussey and M. J. Mahemoff. Safety Critical Usability: Pattern-based Reuse of Successful Design Concepts. In M. McNicol, editor, *4th Australian Workshop on Safety Critical Systems and Software*, pages 19–34. ACS, 1999.

[114] L. M. Hynson, Jr. *Doing Business with South Korea: A handbook for Executives in the Public and Private Sector*. Quorum, New York, 1990.

[115] IBM. *Object-Oriented Interface Design : IBM Common User Access Guidelines*. Que, Carmel, Ind., 1992.

[116] M. Ito and K. Nakakoji. Impact of culture on user interface design. In E. M. del Galdo and J. Nielsen, editors, *International User Interfaces*, chapter 6, pages 105–126. John Wiley & Sons, New York, 1996.

[117] A. Jaaksi. Implementing interactive applications in C++. *Software–Practice and Experience*, 25(3):271–289, March 1995.

[118] M.A. Jackson. *System Development*. Prentice Hall International, Englewood Cliffs, NJ, 1983.

[119] I. Jacobson. The use-case construct in object-oriented software engineering. In J. M. Carroll, editor, *Scenario-Based Design: Envisioning Work and Technology in System Development*. John Wiley & Sons, 1995.

[120] S. C. Jain. *Marketing Planning and Strategy*. South-Western Publishing, Cincinanati, OH, 1993.

[121] B. E. John and D. E. Kieras. Using GOMS for user interface design and evaluation: Which technique? *ACM Transactions on Computer-Human Interaction*, 3(4):287–319, December 1996.

[122] R. E. Johnston. Documenting frameworks using patterns. In *OOPSLA '92*, pages 63–76. ACM, New York, 1992.

[123] R. E. Johnston. Frameworks = (Components + Patterns). *Communications of the ACM*, 40(10):39–42, October 1997.

[124] C. Karat, R. Campbell, and T. Fiegel. Comparison of empirical testing and walkthrough methods in user interface evaluation. In P. Bauersfeld, J. Bennett, and G. Lynch, editors, *CHI 92 Proceedings*, pages 397–404. ACM, New York, 1992.

[125] N. L. Kerth and W. Cunningham. Using patterns to improve our architectural vision. *IEEE Software*, 14(1):53–59, January/February 1997.

[126] T. Kletz. *Plant design for safety: a user-friendly approach*. Hemisphere, 1991.

[127] P. Kogut. Design reuse: Chemical engineering vs. software engineering. In L. Rising, editor, *The Patterns Handbook*, pages 391–406. Cambridge University Press, Cambridge, UK, 1998.

[128] J. Kotula. Discovering patterns: An industry report. *Software–Practice and Experience*, 26(11):1261–1276, November 1996.

[129] W. Kozaczynski and G. Booch. Component-based software engineering. *IEEE Software*, 15(5):34–36, September/October 1998.

[130] G. E. Krasner and S. T. Pope. A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. *JOOP*, 1(3):26–49, August/September 1988.

[131] K. Kreft and A. Langer. The Locale Framework. *C++ Report*, 9(9):58–59,62–63,69, sep 1997.

[132] B. Laurel. *Computers as Theatre*. Addison-Wesley, Reading, MA, 1993.

[133] N. Leveson. Final Report: Safety Analysis of Air Traffic Control Upgrades. http://www.cs.washington.edu/homes/leveson/CTAS/ctas.html. Accessed January 20, 1998., 1997.

[134] N. G. Leveson. *Safeware: System Safety and Computers*. Addison-Wesley, Reading, MA, 1995.

[135] K. Y. Lim and J. Long. *The MUSE Method for Usability Engineering*. Cambridge University Press, Glasgow, 1994.

[136] K. Y. Lim and M. S. Usma. Usability evaluation in the field: Lessons from a case-study involving public information kiosks. In J. Tanaka, editor, *Asia-Pacific Computer-Human Interaction (APCHI) '98 Proceedings*, pages 70–75. IEEE Computer Society, Los Alamitos, CA, 1998.

[137] P. Luo. A human-computer collaboration paradigm for bridging design conceptualization and implementation. *Interactive Systems*, pages 129–147, December 1994.

[138] T.V. Luong, J.S.H. Lok, D. Taylor, and K. Driscoll. *Internationalization: Developing Software for Global Markets*. John Wiley & Sons, USA, 1995.

[139] M. Mahemoff. Multiple-Agent Real-time Clock Objects (MARCO) (A Java 1.1 Application), 1998. http://www.csse.unimelb.edu.au/~moke/mmthesis/.

[140] M. Mahemoff. Critique: (A Java 1.2 Application), 2000. http://www.csse.unimelb.edu.au/ ~moke/ mmthesis/.

[141] M. Mahemoff. MMVCCreator: (A Java 1.2 Application), 2000. http://www.csse.unimelb.edu.au/ ~moke/mmthesis/.

[142] M. Mahemoff. Prescribe: (A Java 1.2 Application), 2000. http://www.csse.unimelb.edu.au/ ~moke/ mmthesis/.

[143] M. J. Mahemoff. Software engineering and human-computer interaction. In P. J. Benda, F. Vetere, and J. B. Fabre, editors, *Proceedings of the 1997 Australia-New Zealand Student Conference in Computer-Human Interaction*, pages 40–43. School of Information Technology, Swinburne University of Technology, Melbourne, Australia, 1997.

[144] M. J. Mahemoff. Incorporating usability in the software design process. In M.A. Sasse and C. Johnson, editors, *Human-Computer Interaction: Interact '97*, pages 686–687. IOS Press (for IFIP), Amsterdam, 1999.

[145] M. J. Mahemoff and L. J. Johnston. Pattern languages for usability: An investigation of alternative approaches. In J. Tanaka, editor, *Asia-Pacific Computer-Human Interaction (APCHI) '98 Proceedings*, pages 25–31. IEEE Computer Society, Los Alamitos, CA, 1998.

[146] M. J. Mahemoff and L. J. Johnston. Principles for a usability-oriented pattern language. In P. Calder and B. Thomas, editors, *OZCHI '98 Proceedings*, pages 132–139. IEEE Computer Society, Los Alamitos, CA, 1998.

[147] M. J. Mahemoff and L. J. Johnston. Software internationalisation: Implications for requirements engineering. In D. Fowler and L. Dawson, editors, *Third Australian Conference on Requirements Engineering (ACRE)*, pages 83–90. Deakin University, Geelong, Australia, 1998.

[148] M. J. Mahemoff and L. J. Johnston. Handling multiple domain objects with Model-View-Controller. In C. Mingins and B. Meyer, editors, *Technology of Object-Oriented Languages and Systems 32*, pages 28–39. IEEE Computer Society, Los Alamitos, CA, 1999.

[149] M. J. Mahemoff and L. J. Johnston. The Planet pattern language for software internationalisation. In *Pattern Languages of Programs 1999 Proceedings*, Monticello, IL, 1999. http://jerry.cs.uiuc.edu/~plop/plop99/proceedings/. Accessed September 5, 1999.

[150] M. J. Mahemoff and L. J. Johnston. Reusing knowledge about users. In D. Zowghi, editor, *Fourth Australian Conference on Requirements Engineering (ACRE)*, pages 59–69. Macquarie University, Sydney, 1999.

[151] M. J. Mahemoff and L. J. Johnston. Brainstorming with generic tasks: An empirical investigation. In C. Paris, N. Ozkan, S. Howard, and S. Lu, editors, *OZCHI 2000 Proceedings*, pages 224–231. CHISIG, Ergonomics Society of Australia, Sydney, 2000.

[152] M.J. Mahemoff. Position statement for Interact 97 workshop on "Integrating software engineering and HCI", 1997.

[153] M.J. Mahemoff. Position statement for Interact 99 workshop on "Pattern languages: Creating a community", 1999.

[154] M.J. Mahemoff. A primer on usability patterns. *Simplicity (Newsletter of CHISIG, Ergonomics Society of Australia)*, 2001. To be published in May, 2001.

[155] M.J. Mahemoff. Weaving high-level and low-level patterns: An extended version of the planet pattern language, 2001. Technical Report 2001/21, Computer Science and Software Engineering Department, The University of Melbourne. http://www.cs.mu.oz.au/ tr_submit/test/cover_db/-mu_TR_2001_21.html.

[156] M.J. Mahemoff and A. Hussey. Patterns for designing safety-critical interactive systems, 1999. Technical Report 1999/25, Computer Science and Software Engineering Department, The University of Melbourne. http://www.cs.mu.oz.au/tr_submit/test/cover_db/mu_TR_1999_25.html.

[157] M.J. Mahemoff, A. Hussey, and L.J. Lorraine. Pattern-based reuse of successful design: Usability of safety-critical systems. In D.D. Grant and L. Sterling, editors, *Australian Software Engineering Conference (ASWEC) 2001*, pages 31–39. IEEE Computer Society, Los Alamitos, California, 2001.

[158] M.J. Mahemoff and L.J. Johnston. A high-level pattern language for software internationalisation, 2001. Technical Report 2001/20, Computer Science and Software Engineering Department, The University of Melbourne. http://www.cs.mu.oz.au/tr_submit/test/cover_db/mu_TR_2001_20.html.

[159] M.J. Mahemoff and L.J. Johnston. Usability pattern languages: The "language" aspect. In M. Hirose, editor, *Human-Computer Interaction: Interact '97*, pages 350–358, Amsterdam, 2001. IOS Press (for IFIP).

[160] J. McDermid and T. Kelly. *Industrial Press: Safety Case*. High Integrity Systems Engineering Group, University of York, 1996.

[161] P. E. McKenney. Selecting locking designs for parallel programs. In J. M. Vlissides, J. O. Coplien, and N. L. Kerth, editors, *Pattern Languages of Program Design 2*, pages 501–531. Addison-Wesley, Reading, MA, 1996.

[162] I. Mclelland, B. Taylor, and B. Hefley. User-centred design principles: How far have they been industrialised? *SIGCHI Bulletin*, 28(4):23–25, October 1996.

[163] T. W. Melaon. *International and Global Marketing: Concepts and Cases*. Irwin/McGraw Hill, Boston, MA, 2nd edition, 1998.

[164] G. Meszaros and J. Doble. A pattern language for pattern writing. In R. Martin, D. Riehle, and F. Buschmann, editors, *Pattern Languages of Program Design 3*, pages 529–574. Addison-Wesley Longman, Reading, MA, 1998.

[165] Microsoft. *The Windows Interface guidelines for Software Design*. Microsoft Press, Redmond, WA, 1995.

[166] R. C. Mill, editor. *Human Factors in Process Operations*. Institution of Chemical Engineers, 1992.

[167] M. A. Miller and R. P. Stimart. The user interface design process: The good, the bad, & we did what we could in two weeks. In *Proceedings of the Human Factors and Ergonomics Society. 38th Annual Meeting*, volume 1, pages 305–309. Human Factors and Ergomics Society, Santa Monica, CA, 1994.

[168] P. Molin and L Ohlsson. The points and deviations pattern language of fire alarm systems. In R. Martin, D. Riehle, and F. Buschmann, editors, *Pattern Languages of Program Design 3*, pages 431–445. Addison-Wesley Longman, Reading, MA, 1998.

[169] M. K. Mooij. *Global Marketing and Advertising: Understanding Cultural Paradoxes*. Sage, Thousand Oaks, CA, 1998.

[170] M. J. Muller, D. M. Wildman, and E. A. White. Taxonomoy of PD practices: A brief practitioner's guide. *Communications of the ACM*, 36(4):26–28, June 1993.

[171] J. Nielsen. Usability testing of international interfaces. In J. Nielsen, editor, *Designing User Interfaces for International Use*, pages 39–44. Elsevier Science Publishers, Amsterdam, 1990.

[172] J. Nielsen. *Usability Engineering*. AP Professional, New York, 1993.

[173] J. Nielsen. International usability testing, 1996. http://www.useit.com/papers/-international_usetest.html. Accessed August 1, 1998.

[174] J. Noble. GOF patterns for gui design. In *Pattern Languages of Program Design 1998*, 1998. http://jerry.cs.uiuc.edu/plop/plop98/final_submissions/. Accessed March 30, 1999.

[175] D. Norman. The 'problem' with automation: inappropriate feedback and interaction, not 'over-automation'. *Philosphical Transactions of the Royal Society of London, Series B*, 327(1241):585–593, 1990.

[176] D. A. Norman. *The Psychology of Everyday Things*. Basic Books, New York, 1988.

[177] D. A. Norman. *The Design of Everyday Things*. Doubleday, 1990.

[178] D. A. Norman. *Things That Make Us Smart*. Addison-Wesley, Reading, MA, 1993.

[179] Open Software Foundation. *OSF/Motif Style Guide: Revision 1.2*. Prentice Hall International, Englewood Cliffs, NJ, 1993.

[180] J. Ostwald. Dynagloss: An extensible glossary of terms. http://Seed.cs.colorado.edu/dynagloss.makeGlossaryPage.fcgi$URLinc=0. Accessed December 12, 2000.

[181] J. Ostwald. Dynasites framework. http://Seed.cs.colorado.edu/dynasites.Documentation.fcgi$URLinc=44&node%=dynasites.doc.home. Accessed December 12, 2000.

[182] J. Ostwald. *Knowledge Construction in Software Development: The Evolving Artifact Approach*. PhD thesis, University of Colorado, Boulder, 1996. http://www.cs.colorado.edu/~ostwald/thesis/home.html. Accessed December 12, 2000.

[183] P. Palanque, F. Paterno, and P. Wright. CHI'98 Workshop (5) on Designing User Interfaces for Safety Critical Systems. ACM SIGCHI Conference on Human Factors in Computing Systems: "Making the Impossible Possible", 1998.

[184] R. D. Patterson. Auditory warning sounds in the work environment. *Philosphical Transactions of the Royal Society of London, Series B*, 327(1241):485–492, 1990.

[185] J. Penrose. Repository of cultural information, 2000. http://titan.it.swin.edu.au/~jp/browse.php3. Accessed December 1, 2000. Developed under the supervision of Lorraine Johnston and with requirements from Michael Mahemoff and Lorraine Johnston.

[186] K. Perzel and D. Kane. Usability patterns for applications on the world wide web. In *Pattern Languages of Program Design 1999 Proceedings*, Monticello, IL, 1999. http://jerry.cs.uiuc.edu/~plop/plop99/proceedings/. Accessed September 18, 1999.

[187] K. Potosnak. When a usability is not the answer. *IEEE Software*, 6(4):105–106, July/August 1989.

[188] L. Prechelt. An experiment on the usefulness of design patterns: Detailed description and evaluation. Technical Report 9/1997, Fakultät für Informatik, 1997. http://wwwipd.ira.uka.de/~prechelt/Biblio/. Accessed December 24, 2000.

[189] R. S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, New York, 4th edition, 1997.

[190] A. R. Puerta. A model-based interface development environment. *IEEE Software*, 14(4):40–47, July/August 1997.

[191] F. Rafii and S. Perkins. Internationalizing software with concurrent engineering. *IEEE Software*, 12(5):39–46, September 1995.

[192] Rational. Rational Rose. http://www.rational.com/products/rose/. Accessed June 5, 2000.

[193] T. Reenskaug. *Working With Objects: The OOram Software Engineering Method.* Manning, Greenwich, CT, 1996.

[194] D. Riehle and H. Züllighoven. A pattern language for tool construction and integration based on the tools and materials metaphor. In J. O. Coplien and D. C. Schmidt, editors, *Pattern Languages of Program Design*, pages 9–42. Addison-Wesley, Reading, MA, 1995.

[195] D. Roberts, D. Berry, S. Isensee, and J. Mullaly. Developing software using OVID. *IEEE Software*, 14(4):51–57, July/August 1997.

[196] G. F. Rogers. *Framework-Based Software Development in C++*. Prentice-Hall PTR, Upper Saddle River, NJ, 1997.

[197] S. Rosenbaum. Usability evaluations versus usability testing: When and why? *IEEE Transactions on Professional Communication*, 32(4):210–216, December 1989.

[198] W.W. Royce. Managing the development of large software systems: Concepts and techniques. In *WESCON Technical Papers*. IEEE Computer Society, Washington, DC, 1970. Reprinted in 9th International Conference on Software Engineering, 1987.

[199] J. Rubin. *Handbook of Usability Testing*. John Wiley & Sons, New York, 1994.

[200] P. Russo and S. Boor. How fluent is your interface? Designing for international users. In S. Ashlund, A. Henderson, E. Hollnagel, K. Mullet, and T. White, editors, *InterCHI 1993*, pages 342–347. IOS Press, Amsterdam, 1993.

[201] M.A. Sasse, M.J. Handley, and N.M. Ismail. Coping with complexity and interference: Design issues in multimedia conferencing systems. In D. Rosenberg and C. Hutchison, editors, *Design Issues in CSCW*, chapter 9, pages 179–195. Springer, London, 1994.

[202] P. M. Senge, A. Kleiner, C. Roberts, R. B. Ross, and B. J. Smith. *The Fifth Discipline Fieldbook: Strategies and Tools for Building a Learning Organization.* Doubleday, New York, 1994.

[203] Y. Shan. Mode: A UIMS for Smalltalk. In N. Meyrowitz, editor, *ECOOP/OOPSLA '90 Proceedings*, pages 258–268. ACM Press, New York, 1990.

[204] B. Shneiderman. *Designing the User Interface*. Addison-Wesley, Reading, MA, 1998.

[205] S.L. Smith and J.N. Mosier. *Guidelines for Designing User Interface Software*. The MITRE Corporation, Bedform, MA, 1986.

[206] C. L. Stimmel. Hold me, thrill me, kiss me, kill me: Patterns for developing effective concept prototypes. In *Pattern Languages of Program Design 1999 Proceedings*, Monticello, IL, 1999. http://jerry.cs.uiuc.edu/~plop/plop99/proceedings/. Accessed September 18, 1999.

[207] P. Sukaviriya and L. Moran. User interface for Asia. In J. Nielsen, editor, *Designing User Interfaces for International Use*, pages 189–218. Elsevier, Amsterdam, 1990.

[208] Sun Microsystems. *Open Look Graphical User Interface Application Style Guidelines*. Addison-Wesley, Reading, MA, 1989.

[209] Sun Microsystems. Java look and feel design guidelines (online version), 1999. http://java.sun.com/products/jlf/dg/index.htm. Accessed December 27, 2000.

[210] A. Sutcliffe. Personal communication.

[211] A. Sutcliffe. *Human-Computer Interface Design*. Macmillan, Basingstoke, UK, 2nd edition, 1995.

[212] A. Sutcliffe and J. Carroll. Generalizing claims and reuse of HCI knowledge. In H. Johnson, L. Nigay, and C. Roast, editors, *People and Computers XIII: Proceedings of HCI '98*, pages 159–176. Springer, London, 1998.

[213] A. Sutcliffe and M. Dimitrova. Patterns, claims and multimedia. In M. A. Sasse and C. Johnson, editors, *Human-Computer Interaction: Interact '99*, pages 329–335. IOS Press (for IFIP), Amsterdam, 1999.

[214] A. Sutcliffe and N. Maiden. The domain theory for requirements engineering. *IEEE Transactions on Software Engineering*, 24(3):174–196, March 1998.

[215] D. Taylor. *Global software: Developing Applications for the International Market*. Springer-Verlag, New York, 1992.

[216] R. N. Taylor and J. Coutaz, editors. *ISE '94 Workshop on SE-HCI: Joint Research Issues*. Springer, Berlin, 1994.

[217] L. Tetzlaff and D. R. Schwartz. The use of guidelines in interface design. In S. P. Robertson, G. M. Olson, and J.S. Olson, editors, *CHI 91 Proceedings*, pages 329–333. ACM, New York City, 1991.

[218] J. Tidwell. Interaction patterns. In *Pattern Languages of Program Design 1998 Proceedings*, Monticello, IL, 1998. http://jerry.cs.uiuc.edu/~plop/plop98/final_submissions/. Accessed March 30, 1999.

[219] J. Tidwell. The case for user interface patterns, 1999. http://www.mit.edu/~jtidwell/-ui_patterns_essay.html. Accessed September 12, 2000.

[220] United Nations University (Institute of Advanced Studies). One web, one language – UNL, 2000. http://www.iai.uni-sb.de/UNL/unl-en.html. Accessed November 12, 2000.

[221] E. Uren, R. Howard, and T Perinotti. *Software Internationalization and Localization*. Van Nostrand Reinhold, New York, 1993.

[222] P. Van Der Linden (Maintainer). Java programmers FAQ. http://java.sun.com/people/linden/intro.html. Accessed November 28, 2000.

[223] M. Van Welie and H. Traetteberg. Interaction patterns in user interfaces. In *Pattern Languages of Programs 2000 Proceedings*, Monticello, IL, 2000. http://monkey.icu.ac.kr/sslab/proceeding/-PLoP2000/ papers/papersIndex.html. Accessed October 27, 2000.

[224] M. Van Welie and H. Traetteberg. Patterns as tools for user interface design. In J. Vanderdonckt and C. Farenc, editors, *International Workshop on Tools for Working with Guidelines*, pages 313–324. Springer, Berlin, 2000.

[225] Vim Development Team. Vim. http://www.vim.org. Accessed March 15, 1999.

[226] J. Vlissides. Patterns: The top ten misconceptions. *Object Magazine*, 7(1):30–33, March 1997.

[227] W. C. Wake. Patterns for interactive applications, 1998. http://jerry.cs.uiuc.edu/plop/plop98/final_submissions/. Accessed June 19, 1999.

[228] WikiWeb Inc. Wikiweb framework. http://www.wikiweb.com. Accessed December 12, 2000.

[229] S. Wilson, P. Johnson, C. Kelly, J. Cunningham, and P. Markopolous. Beyond hacking: A model based approach to user interface design. In J. Alty, D. Diaper, and S. Guest, editors, *HCI '93*, pages 217–231. Cambridge University Press, 1993.

[230] A. Yeo. World-wide CHI: Cultural user interfaces, a silver lining in cultural diversity. *SIGCHI*, 28(3):4–7, July 1996.

[231] W. Zimmer. Relationships between design patterns. In J. O. Coplien and D. C. Schmidt, editors, *Pattern Languages of Program Design*, chapter 18, pages 345–364. Addison-Wesley, Reading, MA, 1995.

[232] F. Zlotnick. *The POSIX.1 Standard: A Programmer's Guide.* Benjamin/Cumming, Redwood City, CA, 1991.