Pattern Languages for Usability: An Investigation of Alternative Approaches

Michael J. Mahemoff and Lorraine J. Johnston
Computer Science Dept.
University of Melbourne
Parkville 3052 Australia
{moke,ljj}@cs.mu.oz.au

Abstract

The best way to ensure usability is to treat human factors as an input to design, rather than merely evaluating prototypes or design documentation. The capability of pattern languages to facilitate the design process, improve communication, and record design philosophies suggests that they may assist the user-centred design process. Researchers have not yet investigated in detail what a pattern language for usability would offer, or how it could be used. This paper explores several alternative conceptualisations of usability-oriented patterns. Patterns of tasks provide high-level detail about tasks users often perform and how they can be supported. User profile patterns help analysts ensure different kinds of users are supported. Relationships between user-interface objects can also be captured by patterns, assisting system implementors by providing information more directly. Patterns of individual user-interface objects, as well as entire systems are also considered, but appear to have more limited application.

1. Introduction

The concept of design patterns has been proposed as one step towards building more usable systems[4, 6]. Design patterns were originally developed in the context of town planning and architecture and have recently been popularised within the software engineering community. The idea is to capture information about frequently-encountered problems and how they are solved. By recording the symptoms and remedies of typical problems, patterns can improve communication among developers and support design reuse. However, a pattern-based design can only be as good as the patterns it is based on, and patterns for human-computer interaction (HCI) can be formulated in several ways. There are patterns of entire systems, user-interface elements, and user types, for instance[6]. The purpose of the present paper is to investigate several candidate pattern types, with a consideration of what they would look like and how they may be used. Section 2 discusses how the patterns paradigm has influenced building architecture and software design, and Section 3 considers how patterns can improve software usability. In Section 4, we present and analyse some candidate pattern formalisms. The alternative approaches are compared in Section 5.

2. Theoretical foundations of patterns

Christopher Alexander feels strongly about the disjointedness of modern architecture. Having observed the architectural structures of traditional cultures, he concluded that they are built according to universal design patterns[1]. These patterns contribute to what Alexander terms “quality-without-a-name”. That is, a quality which cannot be completely described, but is evoked by terms such as “alive”, “whole”, “comfortable”. Alexander contends that modern architects cannot attain this quality because they are too detached from their products, and have consequently lost sight of the universal patterns.

To rectify the situation, Alexander and colleagues recorded 253 design patterns, varying in scope from the organisation of entire nations down to the design of individual rooms. The key attributes of a patterns are:

Name:	A name to identify the pattern.
Context:	The situation(s) where the pattern is relevant.
Forces:	The forces present which may constrain or suggest alternative solutions. When these forces are in tension with one another, the problem is harder to solve and a compromise may be necessary.
Solution:	A solution which resolves, as far as possible, the various forces.

Consider Alexander’s “A Place to Wait” pattern[2]. The context is any situation where people are waiting for something, such as a doctor’s surgery. Two forces are at play: (a) patients must be present when the doctor can see them, and (b) the timing of this event is uncertain. A suggested solution is to draw in people who are not there to wait. One hospital built a neighbourhood playground which doubled as a children's waiting area, so that the young patients felt at ease before their consultation.

Alexander's patterns do not live in isolation. They are described as components of a pattern *language*, such that each pattern is composed of smaller sub-patterns. In other words, the patterns may be arranged in a network, so that an individual pattern leads to several other patterns some of which should also be applied. "A Place to Wait" suggests several ways to engage people who are not waiting, each being a reference to a separate pattern ("Street Cafe", "Opening to the Street"). The process is recursive; each pattern provides some advice as well as a reference to further patterns. This is why Alexander's patterns vary so widely in their granularity.

The software community has recently begun to adopt patterns in many areas. In contrast to abstract design principles, software design patterns are concrete structures which developers can adapt to suit their own needs. In this way, pattern-based design promotes re-use. Because they are based on an underlying principles, pattern languages may also be used to capture the philosophies of great designers and their work[12]. They also "expand people's communication bandwidth" (p.32)[18] by acting as common reference points among developers. Praise for software design patterns is no longer restricted to theoretical postulation, with several case studies demonstrating their positive impact in industry[5, 13].

.

3. Patterns for usability

Evaluation of a prototype alone is unlikely to guarantee usability; successful identification of defects does not mean that the situation can be rectified[11]. Even when a solution is possible, it may be too expensive or impractical to implement[14]. This suggests the need to evaluate design rather than implementation, since problems uncovered earlier are easier to remedy. However, even this approach has its limitations. No matter how successfully a design review identifies usability defects, the system is still likely to be insufficient if no human factors expertise goes into the design in the first place. Usability-oriented techniques must act as an input to design, not just a means of design assessment. At present, the ability to consider usability upfront is mostly an artform practiced by talented individuals. As the field of user-centered design is shifting from a craft to an engineering discipline[15], there is an increasing need for prescriptive techniques which account for the human factor.

Pattern-based design should be considered as a means of steering the field in this direction. Since usability is a tough concept for many developers, the level of design reuse offered by patterns could significantly improve the state of practice. As HCI is such an interdisciplinary field, the benefits of communication offered by patterns are even more valuable than in conventional software projects. In addition, the notion of user-interface design philosophies matches the idea that a pattern language should be based on a coherent underlying philosophy.

Patterns for usability are relevant at two levels of abstraction: higher-level patterns can solve problems relating to user-interface and task support, while lower-level patterns may give advice about detailed design and implementation of usable software. Interestingly, existing work on patterns closely mirrors this dichotomy. Alexander's research emphasises higher-level concepts whereas software design patterns generally address design and coding issues.

Human experience is a primary focus of Alexander's work. He feels that "a building is as much the life that goes on inside as it is the 'shell' which encloses that life" (p.55)[10]. The ideas embodied in "quality-without-a-name" resemble those discussed throughout HCI literature. Just as the grammar of a spoken language guides the sentences which are uttered, so too do the rules of a pattern language shape the kinds of products which are produced. Alexander's patterns form a language—a set of elements whose interconnections are guided by a well-defined grammar. In contrast, the most popular software design pattern texts have been collections of isolated patterns[12]. It will be apparent in the next section why the language property can be so important in terms of usability.

Usability is one attribute which has not featured highly in software design patterns. Certainly, user-interfaces have been a great motivation for pattern collections. Many of Gamma et al.'s patterns are based on drawing tools, for instance. Occasionally, such systems and the patterns they spawned have addressed implementation of features which improve usability, such as reversibility of operations. However, most patterns in this domain are motivated by concerns such as efficiency, simplicity, flexibility, and portability. The importance of software design patterns in promoting usability should nevertheless be taken into account. A usable system must be designed and coded at some stage, and this is where existing work can be helpful. If a high-level pattern advocates a certain user-interface property, it can be supplemented with the implementation issues which arise in supporting that property.

Pattern languages for interactive usability could have various goals and manifest themselves in different forms[4]. While a degree of attention to human factors has been evident in patterns of work process, few pattern languages have explicitly carried the objective of usability. Casaday has presented one possible format for such a language[6], and it seems an appropriate time to investigate in more detail how patterns for HCI can work. Several possibilities are analysed in the next section.

4. A comparison of usability patterns

This section considers candidates for patterns in HCI. Under consideration are four kinds of patterns: (a) tasks, (b) entire systems, (c) user-interface elements, and (d) users. We are really looking at pattern *meta-collections*, rather than specific collections of patterns. *Meta* denotes the fact that the subject of the collections is being considered, rather than the specific patterns and their relationships. A language of buildings with completely different patterns to Alexander et al.'s[2] could nevertheless retain the original format and be used in the same way. These competing languages would be defined by a common meta-collection. *Collection* indicates that not all groups of patterns qualify as languages, as discussed previously.

Each meta-collection description contains a discussion of how it would be used, as well as an accompanying example pattern. Due to space limitations, examples contain only four fields: name, context, forces, and solution. In a real situation, the fields chosen would depend on the meta-collection and the patterns definitions could be more detailed. The descriptions also consider the meaningfulness of pattern inter-relationships.

4.1. 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[16]. Because most stakeholders can understand task descriptions to some degree, it is a powerful communication tool. Task analysis artifacts can act as a common ground for developers, client representatives, and users alike (see [3]). It is important, then, to ensure that tasks are framed appropriately. The patterns concept suggests itself as a logical way of looking at tasks, as well as imparting information about how they may be supported.

Pattern 1, "Open Existing Document", demonstrates the range of levels a task pattern language can 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 Pattern 1, the advice is quite high-level, i.e. 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 task patterns. The rules of a task pattern language guide how tasks fit together. A "View Document" 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 Document" 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 the task and abstracting away the details.

Name:	Open Existing Document.
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.
Forces:	The user is required to identify one document, but there are usually many to choose from.
Solution:	A document's name alone is usually not a sufficient way to identify it, even if the user created (and named) the document themselves. The system should present cues about the document contents to support the decision-making process.

Pattern 1. An Example of a Task Pattern, "Open Existing Document"

4.2. Patterns of users

Developers need to acknowledge individual differences of users such as frequency of use, general experience with computers, and domain expertise[17]. As Pattern 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. Because it 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[7]. If product development is viewed as a marketing-driven activity, then 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 a smoother coordination between marketers and usability specialists.

Name:	Intermediate User, Domain Expert.
Context:	Any system where a domain expert is likely to use the system long enough to make the transition from novice to intermediate.
Forces:	Software should provide substantial assistance to novices, but these features can distract, obstruct, or irritate users as they become more fluent.
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 [8]).

Pattern 2. An Example of a User Profile Pattern, “Intermediate User, Domain Expert”

4.3. Patterns of user-interface elements

A seemingly obvious choice for a pattern meta-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 Pattern 3 shows, the result is 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 working 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.

Name:	Scrollbar.
Context:	User’s working area exceeds the area which can be displayed at one time.
Forces:	The user should be able to view all of the working area. Showing the entire working area at once means that clarity will be lost.
Solution:	Associate a scrollbar with each dimension of the viewable area which is smaller than the corresponding dimension in the working area.

Pattern 3. An Example of a User-Interface Element Pattern, “Scrollbar”

The above discussion suggests that patterns of individual widgets do not work. A more fruitful approach might be a meta-collection of inter-widget relationships. Highly-usable software often reflects clever arrangements of widgets to produce a synergistic effect. The status object, described in Pattern 4, is an arrangement which is straightforward to implement, but can substantially boost learnability.

This meta-collection resides in 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, a compromise is certainly required. Abstract issues such as task support could not be directly addressed. It might be possible to describe how a set of widgets could produce a dialogue for the “Open Existing Document” task (Pattern 1), but it would only apply to a very specific kind of application. Also, this type of pattern would likely be drawn from snapshots of successful arrangements, and since these arrangements may not be closely related to one another, construction of a well-defined language seems like a difficult task.

4.4. 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, coupled with external factors which might constrain the solution[6]. For example, “Airport Passenger” is a pattern for systems requiring efficiency, reliability, and immediate learning. The solution in this case is to standardise components and procedures where possible.

Another way to classify systems is according to their purpose. Systems which manipulate documents, such as word-

Name:	Show Status.
Context:	A system where objects' purposes and usage mechanisms are not entirely obvious by appearance alone.
Forces:	The user's model of the system should be as accurate as possible. This means that an object's appearance should reveal as much about the object as possible. However, users can be overwhelmed by too much information, and visible space is limited anyway.
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[9].

Pattern 4. An Example of a User-Interface Element Arrangement Pattern, "Show Status"

processors, might be represented by something like Pattern 5. 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 throughout this paper, 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 Pattern 5 would imply. "Document-Manipulator" could be split into more specific patterns ("Word Processor", "Drawing Tool"), but substantial redundancy would result, unless these specific patterns were based on lower-level patterns.

5. Conclusions

This paper has presented several alternative approaches to patterns for usability. Most of the meta-collections discussed could be used in some stage of the development lifecycle. Task patterns help task analysts identify actions the user might perform and suggest the appropriate support mechanisms. User profile patterns enable organisations to ensure that their products are meeting the demands of all users. Patterns of user-interface element arrangements can be applied by designers of screen appearance and software

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

Pattern 5. An Example of a System Pattern classified according to its purpose, "Document Manipulator"

architecture. Key usability issues which arise in development of specific kinds of systems can be captured with patterns of systems.

Given that user-centred design is still in a state of infancy, it is important that patterns are generative. Rather than merely capturing information, they should directly facilitate the creation of systems. Of the formalisms presented in this paper, two stand out in this regard:

Tasks With their emphasis on human experience, task patterns most closely resemble Alexander's patterns. As the "Open Existing Document" pattern demonstrated, it is also possible to provide more detailed design information.

User-Interface Element Arrangements The notion of recording successful widget arrangements closely matches the approach taken with many object-oriented design patterns. This style of pattern is not as explicit about usability as task patterns. However, its proximity to existing techniques suggests that it would be a step in the right direction for developers who might otherwise disregard the human factor.

Task patterns are also promising because of the relationship between patterns. Isolated patterns can certainly be useful, but require more expertise and leave gaps. Moreover, as discussed earlier, the ability to suggest rules for inter-relating patterns is an important property. With task patterns, the grammar helps the analyst to envision how the user will interact with the system.

The objective of this paper has been to examine whether or not patterns are appropriate for the development of usable software, and if so, how they can be appropriately utilised. The Section 4 sample patterns offer a concrete reference point. Pattern languages have improved designers' ability to account for quality attributes such as flexibility and reliability. Now it is time to extend the scope of this work to include another important dimension: usability.

References

[1] C. Alexander. *The Timeless Way of Building*. Oxford University Press, New York, 1979.

[2] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel. *A Pattern Language*. Oxford University Press, New York, 1977.

[3] 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.

[4] 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, Jan. 1998.

[5] K. Beck, J. 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, Washington, D.C., 1996.

[6] 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.

[7] P. M. Chisnall. *Strategic Business Marketing*. Prentice Hall International, Hertfordshire, UK, 3rd edition, 1995.

[8] 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.

[9] E. Gamma, R. Helm, R. Johnson, and R. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[10] S. Grabow. *Christopher Alexander: The Search for a New Paradigm in Architecture*. Oriel Press, Northumberland, UK, 1983.

[11] J. Karat and T. Dayton. Practical education for improving software usability. In I. Katz, R. Mack, and L. Marks, editors, *CHI 95 Proceedings*, pages 162–169. ACM, New York, 1995.

[12] N. L. Kerth and W. Cunningham. Using patterns to improve our architectural vision. *IEEE Software*, 14(1):53–59, Jan. 1997.

[13] J. Kotula. Discovering patterns: An industry report. *Software–Practice and Experience*, 26(11):1261–1276, Nov. 1996.

[14] K. Y. Lim and J. Long. *The MUSE Method for Usability Engineering*. Cambridge University Press, Glasgow, 1994.

[15] I. Mclelland, B. Taylor, and B. Hefley. User-centred design principles: How far have they been industrialised? *SIGCHI Bulletin*, 28(4):23–25, Oct. 1996.

[16] K. Potosnak. When a usability **t**est is not the answer. *IEEE Software*, 6(4):105–106, July 1989.

[17] A. G. Sutcliffe. *Human-Computer Interface Design*. Macmillan, Basingstoke, UK, 2nd edition, 1995.

[18] J. Vlissides. Patterns: The top ten misconceptions. *Object Magazine*, 7(1):30–33, Mar. 1997.