# Principles for a Usability-Oriented Pattern Language

Michael J. Mahemoff and Lorraine J. Johnston

*Computer Science Dept.*
*University of Melbourne*
*Parkville, 3052, Australia*
*{moke,ljj}@cs.mu.oz.au*

## Abstract

*High-level usability principles (e.g. "Familiarity") are difficult to apply to specific projects, and style guides providing more detailed instructions are often misinterpreted and inaccessible. An approach to usability based on design patterns enables designers to learn how certain recurring problems can be solved according to high-level principles. This paper summarises a review of the desirable properties advocated by five popular style guides, and discusses how this list has been modified to provide an underlying philosophy which is appropriate for a usability-oriented pattern language. A sample pattern which exemplifies this philosophy, involving iteration through selectable objects, is described.*

**KEYWORDS** Usability engineering, Design techniques, Style guides.

## 1. Introduction

There has been considerable discussion about how to reconcile the gaps between software engineering (SE) and human-computer interaction (HCI). One of the primary ways to smoothly integrate the disciplines is to develop techniques which enable analysts, designers and coders to do their job while obeying relevant design principles.

Typically, style guides, guidelines, and user-interface standards are employed to facilitate interaction design. Contents of such documents range from high-level heuristics ("Reduce the user's memory load") to detailed rules ("Every form must contain an 'OK' button in the lower-right corner"). They have come under criticism on several counts:

- Difficulty of guideline interpretation [8];
- Too simplistic, and not capable of being used effectively by people who are not human factors experts [5];
- Excessive effort required to find relevant sections (e.g. [18]).

Guidelines may be useful for ensuring applications are consistent for a certain platform, company, or user group. Usability goes beyond consistency, however, since software can be consistently unusable. The general-purpose nature of guideline documents means that usability is often addressed in terms of general principles ("Directness", "Simplicity"). These high-level concepts can only be mapped into specific systems by way of sophisticated interpretation. Even when guidelines are more detailed in nature, experienced designers use them in unanticipated ways, misinterpret or ignore advice, and experience difficulty identifying the appropriate guideline for a particular context [8, 24].

There is an opportunity to provide an intermediate perspective, a level between universal design principles and highly-specific details. The design patterns approach, which has been applied successfully by software developers [3], is a convenient way to help developers place abstract design rules in the context of concrete projects they are working on. In this paper, we introduce the concept of design patterns and explain how a well-defined set of HCI principles can contribute to the creation of design patterns (Section 2). In addition, we document our efforts to identify a philosophy embodying a set of appropriate HCI principles, beginning with a review of principles advocated by popular industry style guides. As these texts typically discuss principles in terms of desirable properties, a summary of these properties is presented (Section 3). Since we also have certain goals in mind for a pattern language, such as a view that the human and computer should work in collaboration, some modifications were applied to the properties. These overriding goals and the modified properties are also discussed (Section 4), and are reflected in an example pattern (Section 5). Finally, there is a discussion about how style guides and patterns can be used in tandem, and an assessment of the resultant philosophy which we have outlined (Section 6).

## 2. Design patterns: the need for an underlying philosophy

The design patterns approach, originally developed for town planning and building architecture, considers recurring design problems, analyses the forces that must be resolved, and exposes a design solution which satisfactorily resolves these forces. An example is Alexander et al.'s `North Face` pattern [1]. The pattern applies whenever a building has a north face. The major force here is that the sun will often cast a long shadow behind it, creating an undesirable shadowed region. A solution is to fill the shadowed region with things that do not require sunlight, such as a carport. Thus, the pattern relates to a particular *context*, identifies the underlying *forces*, and offers a *solution*. Once the solution has been applied, a new context arises in which more detailed problems must be solved. Naturally enough, 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 cohesive structure, with higher-level patterns yielding contexts which are resolved by more detailed patterns. The chief benefit of a pattern language is *generativity*. This means that the designer can begin with a certain context, and work through all relevant patterns to generate the design.

For the elements of a pattern language to be coherent, the language should be based on an underlying philosophy. The patterns demonstrate how forces are identified and resolved according to certain principles; in doing so, they are encapsulating a philosophy. Kerth and Cunningham have noted that Alexander identified, valued, and discarded patterns in a process which embodied his architectural philosophy [15]. A language should generate artefacts aligned to the philosophy on which the language is based; the artefact does not have to be derived directly from the philosophy. Gabriel [9] has likened this phenomenon to that of a tennis player learning to hit a ball. The goal — to hit smoothly and powerfully — can be achieved by advising the player to perform an action which has no obvious connection to the goal: aim to hit a point just beyond the ball. Illustrating a philosophy via patterns forces us to look beyond vague truisms like "Help the user learn about the system", and focus instead on the implications such principles hold for the design process.

An architectural philosophy includes notions of space and habitability, whereas a software design philosophy might be more focused on division of functionality among modules. Nevertheless, the fundamental relationship between philosophy, patterns, and concrete designs remains the same, and is just as relevant for usability-oriented patterns. By capturing patterns of tasks, for example, it is possible to consider the problems inherent in supporting these tasks, and the clever solutions which have emerged across various systems. Chapanis and Burdurka [5] have argued that guidelines and standards lead to poor and diverse interpretation because they are often used by people inexperienced in HCI. A pattern language for usability is not a quick fix for such people. It can, however, accelerate their knowledge acquisition by demonstrating how problems, large and small, may be solved according to user-centered design principles. Furthermore, a vocabulary of patterns which consider both interaction and software design would be a boon to projects of an interdisciplinary nature. To identify patterns which promote usability, it is necessary to understand exactly what we mean by usability. That is, a philosophy is required to capture the meaning of usability.

In aiming for a practical pattern language, we started by reviewing style guides for well-known platforms (e.g. Windows 95), toolkits (e.g. OSF/Motif), and design philosophies (e.g. IBM Common User Access). These guides were chosen because they are all intended to be used by practitioners, they are well-known, and they all contain more detailed rules which are based on the high-level properties we are interested in. The style guides were authored by five large software organisations: Apple, IBM, Microsoft, Open Software Foundation, and Sun. What these texts call a "design philosophy" or "design principles" are typically target attributes of the system being developed, hence we refer to them as "desirable properties". The next section is a summary of these properties, according to the guides.

## 3. Desirable properties advocated by traditional user-interface style guides

The first step to surveying the style guides was to identify those which are in wide use. Guidelines for the following platforms, toolkits, or paradigms were chosen:

- Apple Desktop Interface[Apple-87][2],
- IBM Common User Access[IBM-92][14],
- MS Windows 95[MS-95][20],
- OSF/Motif[OSF-93][22],
- Sun Open Look[Sun-89][23].

The desirable properties suggested by each style guide were collected, and grouped into several categories. These categories are listed here, along with the particular style guide properties which they most closely represent. 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)". Every property in the sections of the guides which dealt with design philosophy is discussed here, with some minor exceptions[1].

It is important to note that the properties in the style guides do not all fit perfectly in their categories, e.g. the [MS-95] notion of directness could fit in with either with direct manipulation or familiarity. Thus, it would be misleading to use the summary to ascertain whether a guide addresses a particular property; each guide addresses most or all listed properties in some way. The list should be viewed as a synthesis of the style guides rather than a means of comparing the guides.

**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. 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 property 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 is the topic of 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

[1] [OSF-93] contains sections called "Adopt the user's perspective" and "Avoid common design pitfalls", which are more process-oriented than product-oriented, and have therefore been omitted. Also, two properties which are listed in[Apple-87], but outside the design principles section, have been included because they otherwise seemed unaddressed by this guide ("Modelessness" and "Plain language").

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, partly 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** *Accomodating 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 properties such as consistency, simplicity, familiarity, and visual design. Furthermore, this property implies 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.

## 4. Towards a pattern language for usability: underlying design philosophy

Some of the style guides' desirable properties are reasonably restrictive, such as the notion that the user must always be in control. An alternative view would be that the user and computer work together in collaboration, and both agents share control. Such a perspective can be viewed as an overriding principle, or a goal, of our pattern language. This section identifies the goals of our pattern language and a set of desirable properties, which result from adjusting the style guides' desirable properties in accordance with those goals. Furthermore, if pattern languages are to be of utility to developers, it is important to consider the processes which would facilitate their use on a project. Thus, the patterns are based on a philosophy which is composed of three parts: goals, desirable properties, and process-oriented principles. These will now be discussed in turn.

### 4.1. Goals For a Pattern Language

The pattern language we are developing is driven by several goals. These goals are not only evident in the selection of the desirable properties and processes discussed in the subsequent sections, but also in the patterns themselves.

**1. Usability in Context** Usability can be an elusive concept to define and measure, let alone achieve. To make matters worse, it has often been neglected by developers. However, this does not mean that we should automatically elevate its status beyond that of other software qualities. The relationship among various software attributes is a complex web, and the overall context of development must therefore be kept in mind when usability is considered. Thus, a pattern language for usability should concern itself not only with what the user experiences, but also with matters such as reliability and ease-of-modification. Ultimately, these issues have a powerful impact on users anyway; for example, software which can be rapidly modified will make it easier for developers to respond to change requests. At the same time, certain presumptions sometimes made by developers must also be questioned. Cooper, for instance, has suggested that developers discard "scarcity thinking"—the obsolete perception that resources such as memory and CPU cycles are priceless—and embrace "abundance thinking", i.e. harness these moderately-priced resources in a user-centered way [7].

While some high-level patterns might be considered purely in user-centered terms, a pattern language should ideally address detailed design issues. Otherwise, the language could generate a perfectly usable system which no-one would care to implement.

**2. User and computer in collaboration** Instead of viewing the computer as a passive tool controlled by the human, greater progress can be made by treating the two parties as mutual collaborators. Under this framework, the goal of design is to optimise the system while taking into account the characteristics of each entity. As an example, the machine remembers raw data better than its user, but the human is more capable of making mission-critical decisions. One inspiration for this view is Laurel's work [16], where it is argued that software should treat humans and computers as agents working together in a common arena, rather than being divided by the interface. This concept of co-operation brings into question some of the desirable properties proposed by the industry guidelines; direct manipulation and user control can be counter-productive when the user must perform simple, time-consuming tasks which could just as easily be delegated to the computer[11], such as searching the web for an email address.

The desirable properties for a usability-oriented pattern language should therefore be a consequence of the strengths and weaknesses of the interaction participants, i.e. the computer and the user.

**3. Variety of interaction styles** As alluded to by the previous points, the familiar WIMP-style of interaction should not be treated as invariant. Usability is just as important in other systems: consumer electronic equipment (phones, video recorders), virtual reality devices, technology driven by voice or pen-based input, and command-based systems. Desirable properties of a generic usable system should have a broader base than those which are WIMP-specific. Furthermore, the properties proposed by the style guides focus more on novices than experts, a fact which reflects the origins of the commercial GUI (see [17]). While software should certainly cater for novice users, it is important to realise that experts may require different styles of interaction, for example scripting languages.

### 4.2. Desirable properties of usable systems for a usability-oriented pattern language

As discussed earlier, the properties listed here are the result of analysing the properties suggested by the industry guidelines, and applying a modification process. This process is based mostly on the goals described in the previous section. In constructing each particular property listed here, there are certain style guide properties which have been particularly relevant, even though they might be modified somewhat. For each property, then, the most relevant style guide properties are shown in parentheses. Direct manipulation is the only style guide property which is not accounted for. While certain concepts of direct manipulation might be useful, our goals of supporting user-computer collaboration and diverse interaction styles suggest that it would be inappropriate to treat it as a basis for this philosophy.

**1. Task efficiency (Simplicity, Efficiency)** Task efficiency requires that software assists users of varied experience levels in accomplishing 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 idiosyncracies of the application. Thus, the user is efficient and the activity is also enjoyable.

**2. Reuse (Consistency)** Whereas most standards focus on consistency, a preferable concept is reuse (borrowed from Constantine [6]). 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 Nielson [11] 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.

**3. User-Computer Communication (Feedback)** 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 [7]) and applications in which we see what we get (WYSIWYG), instead of what we need (WYSIWYN [6]). 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 represent 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.

**4. 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 [16] 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.

**5. 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. In keeping with the goal of collaboration, the computer could initiate certain changes as well. In some situations, for instance, it might be appropriate for a computer to reduce output accuracy as a means of speeding up the interaction.

**6. 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.

### 4.3. Process-oriented principles for a usability-oriented pattern language

The style guides generally avoid delving into process issues, perhaps because they assume these decisions are best left to developers. However, for a pattern language to be applied successfully, there must at least be some general assumptions about how it will be used. Hence process-oriented principles are also discussed.

**1. Iteration** When we consider the human as part of the overall system, it becomes apparent that iterative development must occur. That development is necessarily iterative is only partly explained by our inability to predict human behaviour reliably [12]; even if we could foretell precisely how a user would interact with a system, there is an inherent need to iterate because as people use a system, they begin to create new tasks, which the system must then be adapted to support [4]. At the same time, iteration can be expensive and error-prone if not planned correctly. Patterns should therefore ensure that the relevant information is known before design/redesign occurs, and this may involve user consultations and participatory design.

**2. Understand user and usage context** Just as software engineers should acknowledge hardware constraints when designing a system, so too must interaction designers understand the user and the context in which the system will be used.

**3. Emphasis on methods not tools** It is important to realise that software design patterns are not a technique for automatically generating code or even design documentation. Industry experience shows that software design patterns can have a powerful influence on architecture, but tend not to yield large volumes of code [13]. For example, the essence of the model-view-controller pattern is captured in the model, view, and controller class definitions, but the bulk of the code for a particular program would be located in concrete subclasses. Tools, such as heuristics-based user-interface builders, may make invaluable contributions to usability, but patterns for usability should initially address high-level issues such as functionality and architecture.

## 5. Example: A Pattern which Reflects the Design Philosophy

A sample pattern is outlined in Figure 1 to demonstrate how the above philosophy can be encapsulated. There are several potential perspectives a pattern language for usability can take (e.g. patterns of tasks, patterns of users [19]), and the philosophy is intended to apply to any of these conceptions. The example here is a pattern of an interactive activity (it might also be referred to as a task or sub-task, depending on definition). The specific activity in this instance is iteration. In reality, this would be one element of a pattern language, and references to other patterns (which are not described here) have been included (e.g. `Select`).

In combination with other patterns, the `Iterate` pattern will generate designs which conform to the goals, desirable properties, and process-oriented principles discussed in the previous section:

**Goals** The pattern represents a collaboration in that it considers the demands of both the user and computer. Its relevance extends across many kinds of systems, not just WIMP-based GUIs. For the sake of brevity, the pattern does not analyse software design issues, but a more detailed coverage could consider how other patterns may be used to support this task. For example, Gamma et al.'s `Iterator` [10] suggests a flexible object-oriented architecture for providing a class which has elements that can be iterated. Patterns of software architecture, such as Gamma et al.'s patterns, could be used in this way to support functions or user-interface features which are proposed by usability-oriented patterns.

**Desirable properties** Task efficiency is reflected in the attempt to help users easily switch between processing and iterating. Modelessness is suggested to achieve this goal, but it should be clear that modelessness is a means to an end, rather than a goal itself. A suggestion concerning a status object is supposed to give feedback to the user. The idea that the order should be circular provides some robustness, since

**Name** Iterate

**Motivation** The `Select` patterns suggests that one way to select objects is to iterate through them.

**Context** This pattern is applicable wherever the user can select from a finite set of objects.

**Forces**

- The user may be iterating to find a particular object, or to process a sequence of objects one at a time (i.e. process, select next item, process, etc.).
- The user may want to iterate through a subset of the objects, and at the same time be free to jump to an out-of-sequence object.
- The computer must be provided with a way to order the objects. This sequence should be predictable to the user.
- The user should generally be able to start iterating from any object, and this means that the sequence should be circular, i.e. "wrap around". It is usually desirable to iterate forwards and backwards, possibly to let the user `Return to Earlier State`.

**Solution** Consult users to determine the logical sequence for a particular set of objects. Even if the order is not immediately obvious, it should at least be predictable. Try to make the iteration modeless, so that the user can easily switch between processing and iterating, and can also jump to another object easily. This means that the user should be able to `Check Status` of the material they are the working on, to understand where they are in the iteration process. NetDeploy Packer [21], for example, contains a split window which contains the forms which users process on one side, and a task list on the other side. The forms themselves enable users to access next and previous forms. At the same time, a task list which shows the status of each task (e.g. Done, Errors, Alert) is present, and enables users to move to a form which is not in the regular sequence.

**Examples** Editing a sequence of documents prepared by someone else; stepping through months on a calendar; activating "Search next" command; iterating through items on a GUI menu; channel-surfing through television stations; reviewing command-line history.

**Figure 1. An interactive activity pattern, `Iterate`, which encapsulates the design philosophy discussed in Section 4.**

"wrapping around" might be viewed as a form of input tolerance provided that appropriate support is provided (such support is unfortunately scarce in most systems). Comprehensibility is attained by ensuring that users find the ordering logical, or at least predictable. Reuse and flexibility are not explicitly covered, although a more detailed version of `Iterate` could do so.

**Process-oriented principles** The pattern is focused on finding an iteration order which is relevant to users, and as such requires an understanding of the user, as well as allowing for a structured participatory design process. User evaluation of prototypes, however, is not covered in this particular pattern. Finally, the pattern focuses on methods rather than tools, although it certainly does not rule out applications like user-interface builders or other tools.

## 6. Conclusions

This paper has explained why it is helpful to establish a philosophy for usability in order to produce patterns for generating usable software. The desirable properties arising from our investigation of popular style guides have been summarised, the development of a usability philosophy based partially on those properties has been documented, and an example has been provided to demonstrate how a pattern can reflect this philosophy. The overall goals, desirable properties, and processes of this philosophy are generally coherent with one another, and should therefore lead to a well-integrated pattern language. There will need to be some compromises made in determining the detail covered by the patterns, though. For example, it may be difficult to treat software design in detail and simultaneously ensure that the language is applicable for various interaction mechanisms.

The philosophies advocated by style guides are universal. It is easy to conceptualise a term such as "Simplicity", but when it comes to applying this target property to a specific situation, the solution is often unclear. A pattern language consists of elements which represent recurring problems and show how designers have produced appropriate solutions. Sections of style guides dealing with appropriate use of user-interface objects also fail to ensure usability. They are typically concerned with ensuring that components are used in a consistent manner. Some of the mechanisms they prescribe may be classified as usable, but many are nevertheless specific to a certain interaction mechanism (e.g. direct manipulation GUIs), and therefore cannot capture more fundamental issues such as how to support users' tasks, and how to design the resulting software. These concepts will help us transfer existing knowledge to new systems, as well as systems where usability has not often been considered.

This is not to say that style guides are unimportant. Users who work with an operating system should find software more comprehensible if certain features are kept consistent across applications. Look-and-feel should still be prescribed by style guides, while a pattern language can be used to capture the more fundamental design issues.

## References


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

[2] Apple Computer. *Human Interface Guidelines: The Apple Desktop Interface*. Addison-Wesley, Reading, MA, 1987.

[3] 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.

[4] 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.

[5] A. Chapanis and W. J. Budurka. Behaviour and information technology. *Specifying Human-Computer Interface Requirements*, 9(6):476–492, Nov. 1990.

[6] L. L. Constantine and L. Lockwood. Usage-centered design through essential modeling, 1996.

[7] A. Cooper. *About Face: The Essentials of User Interface Design*. IDG Books, Foster City, CA, 1995.

[8] F. de Souza and N. Bevan. The user 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.

[9] R. Gabriel. *Patterns of Software*. Oxford University Press, New York, 1996.

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

[11] D. Gentner and J. Nielson. The anti-Mac interface. *Communications of the ACM*, 37(8):70–82, Aug. 1996.

[12] H. R. Hartson and D. Boehm-Davis. User interface development processes and methodologies. *Behaviour & Information Technology*, 12(2):98–114, Mar. 1993.

[13] R. Helm. Patterns in practice. *SIGPLAN Notices*, 30(10):337–341, Oct. 1995.

[14] IBM. *Object-Oriented Interface Design : IBM Common User Access Guidelines*. Que, Carmel, Ind., 1992.

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

[16] B. Laurel. *Computers as Theatre*. Addison-Wesley, Reading, MA, 1993.

[17] H. Lehtman. What you see is all you get. *Interactions*, 4(1):49, Jan. 1997.

[18] J. Löwgren and U. Laurén. Supporting the use of guidelines and style gudies in professional user interface design. *Interacting with Computers*, 5(4):385–396, Dec. 1993.

[19] M. J. Mahemoff and L. J. Johnston. Pattern languages for usability: An investigation of alternative approaches. In J. Tanaka, editor, *APCHI 98 Proceedings*, pages 25–31. IEEE Computer Society, Los Alamitos, CA, 1998.

[20] Microsoft. *The Windows Interface guidelines for Software Design*. Microsoft Press, Redmond, WA, 1995.

[21] Open Software Associates. NetDeploy Packer v.3.0. http://www.osa.com.au.

[22] Open Software Foundation. *OSF/Motif Style Guide: Revision 1.2*. Prentice Hall International, Englewood Cliffs, NJ, 1993.

[23] Sun Microsystems. *Open Look Graphical User Interface Application Style Guidelines*. Addison-Wesley, Reading, MA, 1989.

[24] L. Tetzlaff and D. R. Schwartz. The use of guidelines in interface design. In S. P. Robertson, G. M. Olson, and J. Olson, editors, *CHI 91 Proceedings*, pages 329–333. ACM, New York City, 1991.