Usability Pattern Languages: the "Language" Aspect

Michael Mahemoff * **& Lorraine J. Johnston** †

* CSSE Dept., University of Melbourne, Parkville, Victoria, 3052, Australia.

† School of IT, Swinburne University, Hawthorn, Victoria, 3122, Australia.

m.mahemoff@csse.unimelb.edu.au, lorraine@it.swin.edu.au

Abstract: The pattern language concept, adapted from building architecture and detailed software design, has recently been applied to HCI by a number of researchers. We argue that the "language" aspect of pattern languages is critical and outline thirteen research efforts in HCI patterns. Many pattern collections have broad scopes, and we argue that this results in patterns which are not as interdependent as a narrow scope would allow. The argument is illustrated with our Planet pattern language, which helps developers reuse knowledge for internationalised software. The narrow scope, namely a focus on the needs of internationalised users, allows us to produce interdependent patterns which range in abstraction level, from organisational process to high-level specification to detailed software design. Thus, the language supports a generative, interdisciplinary, approach to reusing knowledge in HCI.

Keywords: Pattern Languages, Design Patterns, Design Reuse, Internationalisation, HCI, Usability, Software

1 Introduction

Patterns for HCI have recently been developed by a significant number of researchers (e.g. Borchers, 1999; Sutcliffe & Dimitrova, 1999; Tidwell, 1998). There are now several pattern collections which are intended to improve reuse of human-computer interaction. Yet, there continues to be a question about the efficacy of the approach. Some patterns are obvious and others are unproven speculations which defy the term "pattern". It is often difficult to see how a pattern collection could offer true benefits to a practitioner. Our view is that the critical notion of "language" in "pattern language" is all too often overlooked. In this paper, we explain what constitutes a true pattern language and argue that it is pattern languages, and not "pattern collections" or isolated patterns, which will provide the greatest long-term benefits to HCI.

In this paper, we discuss what is meant by patterns and pattern languages, and explain how pattern languages can benefit HCI (Section 2). We argue that a tight focus facilitates a well-integrated set of patterns, and observe that few present approaches do have narrow scopes (Section 3). We then describe the Planet pattern language for software internationalisation, which demonstrates the relationship between scope and language coherence (Section 4).

2 Background: Patterns and Pattern Languages

The design pattern approach was originally developed for town planning and building architecture. Christopher Alexander, an architect who was disaffected with modern architectural practice, argued that a rigid design process had led to the prevalence of impractical solutions. He drew inspiration from ancient cultures, which had evolved buildings and town plans over generations (Alexander, 1964), and noticed there were recurring features, or "patterns". With his colleagues, he published 253 patterns for town planning and building architecture (Alexander et al., 1977), for reuse in new projects.

A pattern has defined fields, including Context, Problem, Forces, and Solution. A pattern is used in a certain design Context, and considers a recurring design Problem in this context. It focuses on the Forces which confront the designer, before describing a Solution—a proposed approach to the situation which resolves the tensions among forces. Consider

Alexander's A Place to Wait pattern (Alexander et al., 1977). 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 is ready, 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 created a neighbourhood playground which doubled as a children's waiting area, so that the young patients felt at ease before their consultation.

An individual pattern can contribute to increased reuse, but the biggest gains arise when patterns are carefully combined. Once the solution to a pattern has been applied, a new context arises in which more detailed problems require solution. Further patterns can be invoked to capture the problem-solving processes inherent in this new context. A *pattern language* is formed when a collection of patterns is arranged into a network of interdependent patterns, especially where higher-level patterns yield contexts which are resolved by more detailed patterns. This allows a designer to apply the pattern language generatively, beginning with a specific context, and working through all relevant patterns to generate the design. In A Place to Wait, the essential solution is to mix 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 propagative nature explains why Alexander's patterns vary so widely in their granularity. The language begins with the distribution of towns (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).

To appreciate the importance of a pattern *language*, it is necessary to comprehend the subjective basis of pattern languages. Far from being the objective and exhaustive catalogue of ideas they may initially seem, patterns are based heavily on an underlying set of values. They explain how forces are identified and resolved according to certain principles; in doing so, they are encapsulating a particular approach. Alexander identified, valued, and discarded patterns in a process which embodied his own architectural philosophy (Kerth & Cunningham, 1997).

To achieve the goal of a usable system, the typical HCI approach is to advocate the use of guidelines. However, such guidelines can conflict with each other, and designers need concrete examples illustrating how to resolve these conflicts. Patterns illustrate how conflicting forces can be resolved in typical design settings. Furthermore, guidelines generally have no inter-relational structure. In contrast, pattern languages aid the designer by beginning with high-level problems and working down to detailed problems. An individual pattern cannot be used in this way. Real-life projects warrant a tightly-related set of patterns, which work together to create a consistent design.

As long ago as 1975, Fred Brooks declared that conceptual integrity was a key issue in system design (Brooks Jr., 1995). By applying closely-related patterns which propagate from one to another, it is possible to achieve this unity of approach.

A good example of a well-integrated pattern language for HCI is provided by Bradac & Fletcher (1998). The language has a very specific focus: design of GUI-based forms. The first of five patterns, 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 a Home Country of USA needs a particular address format, while a user who selects Australia requires a different format. The Address subform then 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.

A designer can approach Bradac et al.'s 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. This makes the language *generative*. Furthermore, a common set of principles lies beneath the patterns. These are implicit and, in this language, relate to the usual high-level principles associated with GUIs (e.g. Recognise Not Recall). The patterns work together to produce systems which adhere to these principles. It would be nonsensical to produce a collection of patterns which are based on incompatible principles.

Pattern languages have mainly been the domain of the architectural software design community. While attributes such as maintainability and reliability are

considered, usability is not often a primary concern. However, unlike a computer program, user reactions cannot be accurately predicted. Usability patterns can document features which worked for users, reducing the costly trial-and-error cycle.

3 Patterns in HCI: Current Research Efforts

In analysing existing research efforts, it is helpful to categorise approaches according to three dimensions:

Level of Abstraction. Possibilities include patterns of user-interfaces, of tasks, of users, and so on.

Target Medium. Possibilities include conventional GUIs, websites, handhelds.

Specialised Requirements. Possibilities include specialised application domains (banking), specialised user characteristics (blind users), specialised software qualities (safety-critical systems)

Several HCI pattern collections have been developed in recent years, in parallel to the present work. Table 3 summarises the best-known contributions, inferring where each lies according to the classification above. In each case, the descriptions of the dimensions have necessarily been simplified.

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 two approaches other than ours (Breedvelt-Schouten et al., Stimmel) relate to various systems, but feature different levels of abstraction: task models, development process.

At present, few pattern collections are tightly constrained to their target medium or specialised requirements. Many have a target medium of GUI applications, and occasional evidence that websites have been considered. However, this is still a very broad category—while the principles of design for GUIs are well-understood, they do vary across platforms. Collections of patterns which are not tightly constrained in some way are unlikely to produce an end product which is conceptually self-consistent. In terms of specialised requirements, few approaches constrain their scope.

There is certainly a benefit in capturing successful design concepts, whatever the format. A large catalogue of HCI patterns would be an excellent resource for students and practitioners alike. But while reusable knowledge repositories are developed, it is important to recognise the importance of the *language* aspect. A pattern language makes generative design possible and contributes significantly to the conceptual integrity of the end product. Furthermore, it is possible that constraining scope in some areas may enable us to expand scope in other areas. We are particularly interested in expanding the levels of abstraction covered by a pattern language. By constraining the target medium or specialised requirements, it should be possible to create patterns which relate high-level concepts to detailed software. Perhaps the closest approach to our work is Borchers' patterns. In this interdisciplinary approach, there are separate languages for software patterns, HCI patterns, and domain-specific patterns—musical patterns in Borchers' example. Our approach differs in that all patterns focus on our area of interest—in this case, software internationalisation. The patterns are highly inter-dependent—they are intended to work effectively with each other, and would have little use in isolation.

Pattern *languages* for HCI may not immediately gain widespread acceptance. They require more effort to construct than general-purpose HCI pattern collections, as each pattern must be consistent with the others, and all must work towards common goals. If they address situations with limited scope, they will not be as broadly applicable. Yet, a prerequisite to a "general-purpose" HCI language is a series of highly-focused pattern languages. Such languages would also be valuable to practitioners working in the target area.

4 Planet: An Example of an HCI Pattern Language

4.1 Background

Planet is our attempt to demonstrate that, by constraining the scope, a rich set of inter-pattern relationships can be captured. The language has a specialised requirement: software internationalisation. This tight focus has enabled us to look at a variety of target media and, more importantly, to address multiple levels of abstraction. The patterns were created by studying the issues involved in software internationalisation and studying successful features of internationalised systems. We have documented the language and a sample application, Critique, which realises many of the patterns (Mahemoff, 2001).

As mentioned, a pattern language is based on

Approach	Level of Abstraction	Target Medium	Specialised Requirements
Tidwell (1998): Interaction Design Patterns	Systems, Multiple & single UI elements, Functionality	GUI Applications, Websites	None
Brighton Usability Group (2001): Brighton Usability Pattern Collection	Entire systems, Multiple & single UI elements, Functionality	GUI Applications	None
Van Welie & Traetteberg (2000): Amsterdam Pattern Collection	Multiple UI elements, Functionality	GUI Applications, Websites	None
Coram & Lee (1996): Experiences	Multiple and single UI elements, Functionality	GUI Applications	None
Wake (1998): Patterns for Interactive Applications	Multiple UI elements, Functionality	GUI Applications	None
Borchers (1999): Interdisciplinary Design Patterns	Both High-level and low-level, Functionality	Various	Can be Domain-Specific
Cybulski & Linden (2000): Multimedia Patterns	Multiple UI elements, Functionality	Multimedia Applications	None
Bradac & Fletcher (1998): Patterns for Form Style Windows	Multiple UI elements	GUI Forms	None
Perzel & Kane (1999): Usability Patterns for Applications on the World Wide Web	Multiple and single UI elements, Functionality	Websites	None
Riehle & Züllighoven (1995): Tool Construction and Integration	Multiple UI elements, Functionality, Software Design	Desktop Applications	Artifact Manipulation
Breedvelt-Schouten et al. (1997): Reusable Structures in Task Models	Tasks	Various	None
Stimmel (1999): Patterns for Developing Prototypes	Development process	Various	None
Mahemoff & Johnston (1999; Mahemoff, 2001): Planet Patterns	Development Process, High-Level Specification, Software Design	Various	Software Internationalisation

Table 1: A Survey of Recent HCI Pattern Collections

an underlying set of principles. Although principles often remain implicit, we now state those of Planet's to illustrate our point.

Developers should Acknowledge Cultural Diversity. Cultures differ in obvious areas such as units of measurement as well as in more subtle areas such as social rules.

A Universal Version is Unrealistic. Instead of seeing cultural diversity as a barrier, designers can exploit the fact that people will have a particularly strong connection with features targeting their own needs.

Every Person has Individual Needs. Cultural differences are important only to the extent they establish all the parameters of the software; individuals should still be free to choose their own values of these parameters.

Developers should Reuse Knowledge about Users. Since the process of learning about foreign cultures is difficult and time-consuming, reusing information saves time and money.

Enable Then Localise For optimal efficiency, the core software components should not be duplicated.

4.2 Language Structure

As Figure 1 shows, the Planet language consists of Patterns at three levels of abstraction:

- **Organisational process patterns** help an organisation track information about the cultures their software supports.
- **High-Level Specification Patterns** guide decisions regarding functionality, user-interfaces, and

configuration of preferences.

- **Detailed Design Patterns** support detailed software design, at a similar level to conventional software design patterns.

The patterns are structured so that they can be used generatively: high-level patterns lead to more detailed patterns. This large variation in abstraction level is unusual for an HCI language, but has precursors in work such as Alexander's patterns (Alexander et al., 1977).

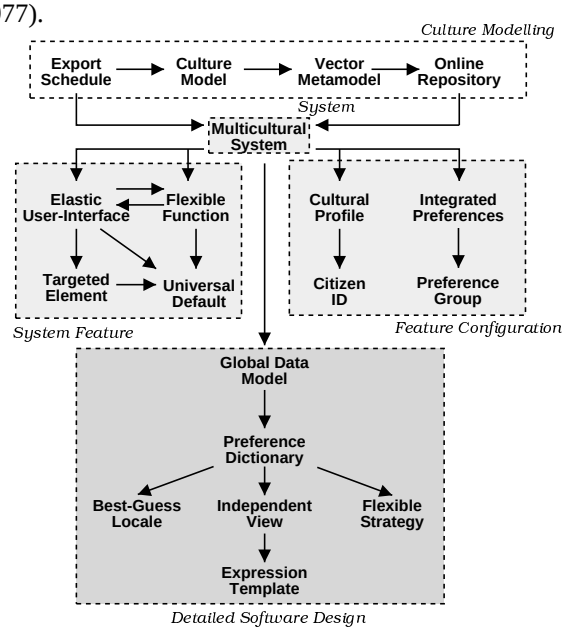

Figure 1: Map of Planet Pattern Language, showing three layers: organisational patterns, high-level specification patterns, detailed design patterns.

4.3 Planet's Patterns

In this section, we show the core solution of each pattern. Referencs to other patterns are shown in `Typewriter` font. First, the organisational patterns:

Export Schedule: Produce a schedule which indicates *when* each target culture will be supported, and *how important* it is to support each target culture.

Culture Model: Construct models of cultures which are relevant to your projects. When you discover new information about a culture, add it to the culture model.

Vector Metamodel: Determine the dimensions of cultures that interest you, and characterise each `Culture Model` as a vector with a value for each dimension.

Online Repository: Create an online repository accessible organisation-wide. Use `Culture Models` all from the same `Vector Metamodel`.

Next are the high-level specification patterns. These begin with a meta-pattern, `Multicultural System`, which serves to structure the overall language:

Flexible Function: 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.

Elastic User-Interface: Design the overall structure for the user-interface flexibly, so that UI elements can subsequently be redefined and rearranged without massive design changes.

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

Universal Default: For each culture-dependent feature in the target system, make a default which is universally meaningful.

Cultural Profile: Provide a default profile for each target culture, a profile which specifies the value of each culture-dependent feature.

Citizen ID: Determine the user's culture on first use and permit this preference to persist. Select the `Cultural Profile` based on the user's Citizen ID.

Integrated Preferences: Integrate culture-related preferences with general preferences.

Preference Group: Group related preferences, so the user must make only one choice to set all preferences in the group to logically-related values.

A small set of patterns can be used to drive the detailed design for systems specified according to the patterns above. This set of patterns constitutes the detailed software patterns of Planet:

Global Data Model: Encapsulate data required to support all cultures in a global data model, independent of the user-interface.

Preference Dictionary: Encapsulate all of the user's current preferences, whether culture-specific or not, in a single dictionary (i.e. key-value pairs) class.

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 to override the global `Preference Dictionary`.

Independent View: Create one or more views of the `Global Data Model`, and store the user's preferred view within the `Preference Dictionary`.

Expression Template: Encapsulate each culture-specific expression in a template string. Store the template string in the culture's `Preference Dictionary`.

Flexible Strategy: Create an abstract class which declares the interface for the algorithm, then create culture-specific implementations of this class (based on Gamma et al.'s (1995) `Strategy` pattern).

4.4 Examples of Patterns

In this section, a sketch is provided for one pattern in each of the three levels of abstraction.

Organisational Pattern: Online Repository

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

Problem: How can a collection of culture models be organised to be useful for software projects?

Forces: (a) Organisation-wide Culture Models avoid duplication; it is feasible and desirable to transfer information learnt from one project to other projects. (b) Information about cultures is often discovered in physically distant locations. (c) If developers cannot access models quickly and easily, the information will be ignored. (d) If developers cannot update models easily, the information will lose accuracy over time. (e) Developers may wish to look up a specific Culture Model, but they may also wish to explore information in other ways, e.g. comparing two cultures, or considering a single factor across numerous cultures.

Solution: Create an online repository for 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: (a) Provide browsing facilities which present each culture and factor. (b) Provide facilities to search the `Culture Models`. (c) Link from one model to another if it helps to demonstrate a point of similarity or difference. (d) 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: (a) Facilitate discussion among contributors, e.g. via a mailing list or within the repository system. (b) Make one individual responsible for managing the overall repository, promoting the repository within the organisation. (c) Make one person responsible for maintaining each `Culture Model`.

Examples: Fernandes (1995) contains some tables showing factors versus culture. However, the text stops short of exhaustively listing this information; the cultures listed varies according to the factor.

Ito and Nakakoji have prototyped a system for retrieving culture-specific details (Ito & Nakakoji, 1996).

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`.

High-Level Specification Pattern: Cultural Profile

Context: You are designing a `Multicultural System`.

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

Forces: (a) A `Multicultural System` offers many choices because each feature has several culture-specific variants, all of which exist in a single version. Configuring can be time-consuming for users. (b) Most users want to begin working on a product right away, rather than exert effort configuring it. (c) Users may not be capable of specifying the appropriate settings for some variables, even in their own country. Imagine asking a user to state 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 are able to begin working, and can tweak settings later 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 (Grosjean, 1982). 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.

Many websites of multinational companies produce different pages depending on the country of origin (e.g. Dell, 2001). Information such as product pricing and local offices are tailored to the specified locale.

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

Detailed Design Pattern: 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 track those parameters which a user can change?**

Forces: (a) `Culture Models` will change as developers learn more and as 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. (b) The user can tailor preferences to their own selection, so culture alone is an inadequate specification of the current "preferences". (c) This information will be used by 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) class. Each parameter, whether or not culture-specific, has a defined key. The preference dictionary can be shared 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, a suitably flexible mechanism for your programming language must be adopted.

Since some preferences may form a `Preference Group`, this may be a recursive class. You may have a message preference dictionary inside a global preference dictionary.

Keys for the **`preferenceBundle`** dictionary
- BackgroundColor
- FontFamily
- MessageBundle
- EvaluationBundle
- NumberFormat

(The `preferenceBundle` object is a member of the `ResourceBundle` class)

Figure 2: `Preference Dictionary` Example: Critique's `preferenceBundle` has key-value pairs for all user-changeable parameters, culture-specific or not.

Examples: Java's `ResourceBundle` class is used by Critique for all preferences. The Preferences Dialog (`PreferenceDialog` class) 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 2).

In Java, the default values are also specified in the `ResourceBundle`. Note there is an `EvaluationBundle` and `MessageBundle`. These are nested `ResourceBundle`s.

The `vim` text editor 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.

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 to the functionality performed, create `Flexible Strategies`. Complement your `Global Data Model` with `Independent Views`.

5 Discussion

HCI patterns research is still relatively new, and researchers have been debating the basic concepts of patterns. Many existing pattern collections do not exploit the "language" property of patterns to a large degree. Pattern languages, while they are less simple to produce, provide a way to achieve the design goal of conceptual integrity, thereby producing a more consistent user interface, and better support for reuse and maintainability.

Planet demonstrates what we mean by a pattern language. To create a tightly-connected language, we constrained our scope to focus specifically on supporting international audiences. The language is based on an explicit set of principles, with the patterns guiding the developer in a direction which adheres to these principles. For instance, the patterns guiding preference configuration show how to consider cultural factors ("Designers should Acknowledge Cultural Diversity"), while keeping the settings flexible ("Every Person is an Individual"). The narrow scope means that a broad range of issues could be covered. Organisational patterns facilitate the ongoing process of learning about cultures and documenting them. Once an organisation has identified its target cultures and sufficiently investigated their needs, high-level specification can proceed. The patterns at this next level address software functionality, user-interface design, and configuration of preferences. Detailed design patterns follow from the high-level specification.

Pattern languages fulfill many of the goals of HCI. Because they can cross levels of abstraction,

they can facilitate a more integrated, interdisciplinary approach. This interdisciplinary approach is also evident in their concrete nature, which makes them approachable for non-HCI specialists, including end-users.

Pattern languages also support an iterative design process: a well-integrated language allows software to be developed with some patterns, then improved with more patterns from the language at a later date. The compromise is a more limited scope; the patterns are not applicable to all situations. Whether a more general language can be created which still has a strong sense of connection among patterns is an open research topic. In the meantime, most organisations create applications which are very similar to one another. They could benefit by capturing their own patterns at differing levels of abstraction, and using them as a basis for interdisciplinary work and ongoing process improvement.

References

Alexander, C. (1964), *Notes on the Synthesis of Form*, Harvard University Press, Cambridge, MA.

Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I. & Angel, S. (1977), *A Pattern Language*, Oxford University Press, New York.

Borchers, J. O. (1999), Designing Interactive Music Systems: A Pattern Approach, *in* H. Bullinger & J. Ziegler (eds.), *8th International Conference on Human-Computer Interaction*, Lawrence Erlbaum Associates, London, pp.276–280.

Bradac, M. & Fletcher, B. (1998), A Pattern Language for Developing Form Style Windows, *in* R. Martin, D. Riehle & F. Buschmann (eds.), *Pattern Languages of Program Design 3*, Addison-Wesley Longman, Reading, MA, pp.347–357.

Breedvelt-Schouten, I. M., Paternó, F. & Severijns, C. A. (1997), Reusable Structures in Task Models, *in* H. D. Harrison & J. C. Torres (eds.), *Design, Specification and Verification of Interactive Systems*, Springer, New York, pp.225–238.

Brighton Usability Group (2001), "The Brighton Patterns Collection". Maintained by Griffiths, R. N. at http://www.it.bton.ac.uk/cil/usability/patterns/. Accessed February 18, 2001.

Brooks Jr., F. P. (1995), *The Mythical Man-Month*, 20th anniversary edition, Addison-Wesley.

Coram, T. & Lee, J. (1996), Experiences — A Pattern Language for User Interface Design, *in Pattern Languages of Program Design 1996 Proceedings.* http://www.maplefish.com/todd/papers/experiences/Experiences.html. Accessed August 5, 1999.

Cybulski, J. & Linden, T. (2000), Composing Multimedia Artifacts for Reuse, *in* N. Harrison, B. Foote & H. Rohnert (eds.), *Pattern Languages of Program Design 4*, Addison-Wesley Longman, pp.461–488.

Dell (2001), "Dell Computer Website". http://www.dell.com. Accessed February 18, 2001.

Fernandes, T. (1995), *Global Interface Design*, AP Professional, Chesnut Hill, MA.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995), *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA.

Grosjean, J. (1982), *Life with Two Languages*, Harvard University Press, Cambridge, MA.

Ito, M. & Nakakoji, K. (1996), Impact of Culture on User Interface Design, *in* E. M. del Galdo & J. Nielsen (eds.), *International User Interfaces*, John Wiley & Sons, New York, chapter 6, pp.105–126.

Kerth, N. L. & Cunningham, W. (1997), "Using Patterns to Improve Our Architectural Vision", *IEEE Software* **14**(1), 53–59.

Mahemoff, M. (2001), "Weaving High-Level and Low-Level Patterns: An Extended Version of the Planet Pattern Language". Technical Report 2001/21, CSSE Dept., University of Melbourne. http://www.cs.mu.oz.au/tr_submit/test/cover_db/mu_TR_2001_21.html.

Mahemoff, M. J. & Johnston, L. J. (1999), The Planet Pattern Language for Software Internationalisation, *in Pattern Languages of Programs 1999 Proceedings*, Monticello, IL. http://jerry.cs.uiuc.edu/~plop/plop99/proceedings/. Accessed September 5, 1999.

Perzel, K. & Kane, D. (1999), Usability Patterns for Applications on the World Wide Web, *in Pattern Languages of Program Design 1999 Proceedings*, Monticello, IL. http://jerry.cs.uiuc.edu/~plop/plop99/proceedings/. Accessed September 18, 1999.

Riehle, D. & Züllighoven, H. (1995), A Pattern Language for Tool Construction and Integration Based on the Tools and Materials Metaphor, *in* J. O. Coplien & D. C. Schmidt (eds.), *Pattern Languages of Program Design*, Addison-Wesley, Reading, MA, pp.9–42.

Stimmel, C. L. (1999), Hold Me, Thrill Me, Kiss Me, Kill Me: Patterns for Developing Effective Concept Prototypes, *in Pattern Languages of Program Design 1999 Proceedings*, Monticello, IL. http://jerry.cs.uiuc.edu/~plop/plop99/proceedings/. Accessed September 18, 1999.

Sutcliffe, A. & Dimitrova, M. (1999), Patterns, Claims and Multimedia, *in* M. A. Sasse & C. Johnson (eds.), *Human-Computer Interaction: Interact '99*, IOS Press (for IFIP), Amsterdam, pp.329–335.

Tidwell, J. (1998), Interaction Patterns, *in Pattern Languages of Program Design 1998 Proceedings*, Monticello, IL. http://jerry.cs.uiuc.edu/~plop/plop98/final˙submissions/. Accessed March 30, 1999.

Van Welie, M. & Traetteberg, H. (2000), Interaction Patterns in User Interfaces, *in Pattern Languages of Programs 2000 Proceedings*, Monticello, IL. http://monkey.icu.ac.kr/sslab/proceeding/PLoP2000/-papers/papersIndex.html. Accessed October 27, 2000.

Wake, W. C. (1998), "Patterns for Interactive Applications". http://jerry.cs.uiuc.edu/plop/plop98/final˙submissions/. Accessed June 19, 1999.