Is This a Pattern?

Tiffany Winn and Paul Calder, *Flinders University of South Australia*

Within a given domain, what might appear to be very different problems are often the same basic problem occurring in different contexts. A software design pattern identifies a recurring problem and a solution, describing them in a particular context to help developers understand how to create an appropriate solution. Patterns thus aim to capture and explicitly state abstract problem-solving knowledge that is usually implicit and gained only through experience.

"Pattern" is an often misused buzzword, perhaps because patterns do not lend themselves to prescriptive, formal definitions. To help software designers understand, use, and write better patterns, the authors propose a set of essential characteristics that can serve as a test for "pattern-ness."

Developers can use that knowledge to solve what appears to be a new problem with a tried-and-true solution, thus improving the design of new software.

Recently, the word "pattern" has become a buzzword, and the implicit definition of the pattern concept has become less precise. Defining patterns is tricky, because they are not bound by prescriptive formal definitions. Rather, it is consensus about the existence of particular patterns in a range of existing software that validates them. Yet we still need to develop our understanding of patterns at a theoretical as well as practical level if we are to identify them, use them well, and distinguish them from similar-seeming nonpatterns that are described in a pattern-like style.[1] We must address this lack of clarity if the pattern concept is to retain its force.

For example, many authors agree that Mediator (an object acting as a go-between for communication between other objects) is a pattern.[1,2] But what about algorithms such as Bubblesort and programming techniques such as Extend by Subclassing? On a larger scale, could idiomatic styles of system organization such as Pipe and Filter (pipes connect filters; filters read data from input streams, transform it, and produce data on output streams)[3] represent patterns? And what about activities other than program or system design? For example, HotDraw patterns tell HotDraw framework users how to assemble HotDraw components to construct a drawing editor[4]—but are they really patterns?

We propose a list of essential characteristics of patterns. Such a list cannot provide a definitive test for pattern-ness: given a pattern-like entity that exhibits the essential characteristics, we cannot say that it is defi-

0740-7459/02/$17.00 © 2002 IEEE

Using a dress pattern to make a dress is like using a design pattern to write a piece of software.

nitely a pattern. However, we suggest that any entity that does not exhibit any one or more of the essential characteristics is not a pattern.

What, then, are the essential characteristics of patterns? We have identified nine, each of which is underpinned by the premise that patterns are generative. Architect Christopher Alexander explains,

> *Once we understand buildings in terms of their patterns, we have a way of looking at them which makes all buildings, all parts of a town similar. … We have a way of understanding the generative processes which give rise to these patterns.*[5]

> *[A pattern] is both a process and a thing; both a description of a thing which is alive, and a description of the process which will generate that thing.*[5]

In other words, a pattern does more than just showcase a good system's characteristics; it teaches us how to build such systems.

1. A pattern implies an artifact

Understanding a pattern means having some sort of picture of the "shape" of the potential artifacts being described. For a piece of software, understanding its shape means understanding

- at the big-picture level, how the software works; and
- at the design level, the relationships that the software attempts to capture.

James Coplien put it this way:

> *I could tell you how to make a dress by specifying the route of a scissors through a piece of cloth in terms of angles and lengths of cut. Or, I could give you a pattern. Reading the specification, you would have no idea what was being built or if you had built the right thing when you were finished. The pattern foreshadows the product: it is the rule for making the thing, but it is also, in many respects, the thing itself.*[6]

Explaining how to make a dress by specifying a scissors' route through a piece of cloth is like telling programmers how to write a program by handing over a piece of assembly code. The assembly code might solve the problem, but is unlikely to give them any idea of what they are building. Nor does it give them a means to evaluate their solution's correctness or usefulness. All they can do is rote-copy the given assembly code or work in a higher-level language and compare the assembly code produced with that suggested.

Using a dress pattern to make a dress is like using a design pattern to write a piece of software. The design pattern does not just show how to create the code at a line-by-line level. It also captures the program's key overall structure at a higher level, in a more physical or spatial sense.

Coplien's dress pattern example, flow charts and other graphical representations of standard algorithms, and the structural diagrams provided in pattern catalogs[7] all highlight the important role of pictures in providing big-picture understanding. Algorithms are often best explained with a combination of text, sample code, and pictures. In the case of Bubblesort, for example, a picture can highlight "lighter" elements "bubbling up" and "heavier" ones "sinking down" as the sort operates. Having gained such big-picture understanding, programmers can better adapt sample code to their specific needs, instead of needing to literally copy or translate the given sample code to use it.

In this respect, software patterns are the same as Alexander's architectural patterns. If a proposed software pattern cannot be drawn, it does not embody a physical understanding of a software artifact's structure and therefore is not a pattern.

2. A pattern bridges many levels of abstraction

A pattern is neither just a concrete, designed artifact nor just an abstract description. Rather, it incorporates design information at many abstraction levels, from sample code to big-picture structure diagrams. A pattern facilitates the progression from a vague idea of "I need some software to do this kind of task" to the actual software itself. It also facilitates standing back from a piece of software and analyzing it at more general levels of design. So, a pattern bridges different abstraction levels and thinking about a problem and its solution.

Robert Floyd illustrates what it means to bridge, or link, different abstraction levels in the context of teaching programming:

If I ask another professor what he teaches in the introductory programming course, whether he answers proudly "Pascal" or diffidently "FORTRAN," I know that he is teaching a grammar, a set of semantic rules, and some finished algorithms, leaving the students to discover, on their own, some process of design. Even the texts based on the structured programming paradigm, while giving direction at the highest level, what we might call the "story" level of program design, often provide no help at intermediate levels, at what we might call the "paragraph" level.[8]

Linking different abstraction levels means helping designers make connections between different design levels, such as the story, paragraph, sentence, and word levels. For a particular problem, you could include a general overview at the story level, a flowchart detailing control flow at the paragraph level, algorithms at the sentence level, and sample code at the word level. The flowchart and algorithm, for example, work together to help link idea with implementation, and general overview with sample code. In Floyd's case, he teaches what he calls a standard paradigm for interactive input—prompt-read-check-echo—together with relevant algorithm and sample code, rather than simply providing sample code and leaving students to work out the general paradigm themselves.

Design aids such as patterns should bridge different design levels because a designer's understanding of a problem evolves as the solution develops:

The most common information needs in the early stages of development are ill-defined—users don't know how to solve a problem or where to look for a solution. … As the design unfolds, the designer's understanding of the problem and potential solutions improves, and he refines and elaborates the problem definition until a satisfactory design emerges.[9]

[The designer's] information needs change as the problem context evolves.[9]

It is important, therefore, to develop design aids that help people move gradually from an initial, general understanding of a problem to a more in-depth one. Further, that ability to link different levels of thinking about a design is critical to knowledge reuse, and knowledge reuse is a key to good design.

The challenge in software reuse is not so much to do more of it, but to recognize which reuse is worth doing: "The challenge in reusability is to express the fragmentary and abstract components out of which complete programs are built."[10] All designers, whether consciously or unconsciously, reuse knowledge by learning from their own and others' experience. Design patterns facilitate knowledge reuse by capturing implicit and abstract knowledge in a form that lets a range of people share and use it. But designers also need to link abstraction levels to reuse knowledge. They need to recognize and abstract from useful similarities, at possibly any abstraction level, between their own context and another's.

Floyd said it like this:

I believe it is possible to explicitly teach a set of systematic methods for all levels of program design, and that students so trained have a large head start over those conventionally taught entirely by the study of finished programs.[8]

[You should] identify the paradigms [patterns] you use, as fully as you can, then teach them explicitly. They will serve your students when Fortran has replaced Latin and Sanskrit as the archetypal dead language.[8]

Design patterns do not just present finished programs. They also identify the paradigm or pattern underlying that program. This facilitates software designers' recognition of the pattern and hence their ability to abstract from their own and other contexts and reuse knowledge.

The challenge in software reuse is not so much to do more of it, but to recognize which reuse is worth doing.

3. A pattern is both functional and nonfunctional

Functional issues deal with possibility; they determine what decisions could be (or were) made in a particular context. Nonfunctional issues deal with feasibility; they address a particular decision's desirability in a particular context, or the reasons the decision was made. Nonfunctional issues are critical to good design because they help designers balance conflicting forces, and they facilitate design adaptation in the face of change.

Design often involves balancing related and conflicting forces. For example, you might sacrifice software readability for efficiency. So, good design requires more than understanding just the forces involved. It also requires understanding the relationships

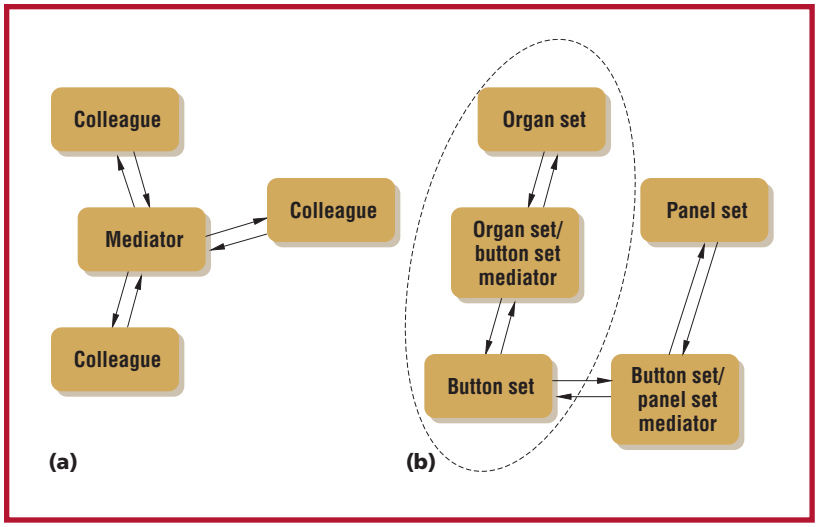

Figure 1. (a) The Mediator pattern; (b) a mediator and two colleagues (circled) in the Prism system.

between those forces. If those relationships are not explicitly documented, that understanding can be lost, with significant consequences. For example, Douglas Schmidt says that the loss of understanding about conflicting forces "deprives maintenance teams of critical design information, and makes it difficult to motivate strategic design choices to other groups within a project or organization."[11] In contrast, explicitly documenting the rationale for design decisions opens that rationale to criticism, thus facilitating improvement in the design process.

Software design is complex and subject to frequent change. Where design is complex, a designer will unlikely be able to grasp the interplay between forces involved simply by observation or intuition. Designers must therefore develop skills, methods, and tools that help clarify that interplay. In a climate where change is frequent and inevitable, an explicit understanding of the reasoning behind design decisions facilitates understanding of the consequences of change.[12] This understanding is critical for effective software maintenance and adaptation.

Design patterns address both functional and nonfunctional design issues. A pattern is inherently functional because it documents a solution to a problem. It is also nonfunctional because it discusses the feasibility of the solution it documents. Patterns address functional design issues by providing fragments of sample code and diagrams of software structure, and by discussing implementation issues. They highlight nonfunctional issues in many ways. For example, a pattern includes discussion of its applicability. For instance, Schmidt's Reactor pattern "explains precisely when to apply the pattern (e.g., when each event can be processed quickly) and when to avoid it (e.g., when transferring large amounts of bulk data)."[11]

Although both functional and nonfunctional design issues are important, the interweaving of the two in discussion is what makes patterns so effective. The functional aspects provide a good solution in a given context; the nonfunctional aspects let us more effectively adapt that solution to another context.

4. A pattern is manifest in a solution

Where a pattern has been used, either consciously or unconsciously, to solve a particular problem, that pattern will be present and recognizable in the developed solution. Although a pattern does capture an abstract idea, it is not just that. It is also the recognition of the generality of that abstract idea, but explained, understood, and demonstrated in a concrete artifact. A pattern is thus not simply a tool used in a program's design and then forgotten. It leaves an indelible mark on the finished product because it focuses on both design process and design structure: it is "both a process and a thing."[5]

For example, a pattern's mark is evident in the Prism system for planning radiation treatment programs for cancer patients.[2] Prism's design addressed the problem of developing a highly integrated but easily extendable computer system. Previous systems that needed to be extendable had often been designed to be loosely coupled, because tightly coupled systems were seen as too complex to extend and their behavior too complex to verify. Yet, in some environments—particularly those using integrated systems—a loosely coupled system does not model the tight real-world coupling between the system's different parts and is therefore difficult and time-consuming to use.

Use of the Mediator pattern makes a system such as Prism possible. The pattern structures integrated systems as "collections of visible, independent components integrated in networks of explicitly represented behavioral relationships."[2] In Prism's case, a set of tumors, a set of corresponding buttons, and a set of panel displays are all independent object sets kept consistent by mediators. The pattern's manifestation—the mediator objects—is clearly visible. In fact, it is critical to the system's overall structure, as Figure 1 shows.

All design approaches strive for the same end: the creation of well-designed software. But approaches that focus solely on design process or methodology do not necessarily leave any identifiable imprint. In contrast, approaches such as design patterns that focus on both design process and structure directly influence the product's visible structure.

5. A pattern captures system hot spots

Software systems must remain stable in a highly dynamic environment. They will frequently both change and be subject to external changes. Building a stable software system is not about foreseeing every possible modification. Rather, stability is about understanding a domain well enough to build a system that can evolve appropriately. As Terry Winograd and Fernando Flores said,

> *The most successful designs are not those that try to fully model the domain in which they operate, but those that are "in alignment" with the fundamental structure of that domain, and that allow for modification and evolution to generate new structural coupling. As observers (and programmers), we want to understand to the best of our ability just what the relevant domain of action is. This understanding guides our design and selection of structural changes, but need not (and in fact cannot) be embodied in the form of the mechanism.*[13]

Central to any pattern is an invariant that solves a recurring problem. But any implemented solution varies or evolves with time. Patterns facilitate good design by capturing what Wolfgang Pree calls system "hot spots"[14]—those parts of a solution likely to change as a developed system evolves. In effect, the pattern captures the invariant and hot spots and provides a structure to manage the interaction between these stable and changing system elements. That structure is critical, because for the invariant part of a system to continue to be invariant in a dynamic environment, the interaction between the invariant and the rest of the system must be carefully defined.

In the software domain, patterns isolate expected invariant system elements from the effects of changes to system hot spots. For example, the Composite pattern deals with situations where treating objects and compositions of objects uniformly is desirable. The invariant is the way objects are structured; the variant is the operation to be performed on the object. Interaction between system elements can be managed by having a Component class, which can have both Leaf and Composite subclasses. Differences between and changes to the Leaf and Composite classes are hidden from their users through the Component class interface.[7]

6. A pattern is part of a language

Every pattern is connected to and shaped by other patterns. Patterns, therefore, are part of a network of interrelated patterns: a pattern language. Alexander explains:

> *No pattern is an isolated entity. Each pattern can exist in the world, only to the extent that it is supported by other patterns: the larger patterns in which it is embedded, the patterns of the same size that surround it, and the smaller patterns which are embedded in it.*
>
> *This is a fundamental view of the world. It says that when you build a thing you cannot merely build that thing in isolation, but must also repair the world around it, and within it, so that the larger world at that one place becomes more coherent, and more whole.*[15]

Alexander's architecture patterns are ordered from the largest patterns, for regions and towns, through neighborhoods, clusters of buildings, and so on, down to construction details. When the patterns are combined, they form a language for describing design solutions. An Accessible Green, for example, can be embedded in an Identifiable Neighborhood. It should also help to form Quiet Backs and must contain Tree Places.

Doug Lea explains the relationship between patterns at different levels of such a pattern language:

> *Patterns are hierarchically related. Coarse grained patterns are layered on top of, relate, and constrain fine grained ones. These relations include, but are not restricted to various whole-part relations. … Pattern entries are arranged conceptually as a language that expresses this layering.*[16]

The software patterns in *Design Patterns*[7] could also be part of a pattern language. For example, an Abstract Factory could be implemented using Factory Method, which in turn could use Template Method to avoid subclassing.

Pattern languages are critical because they capture, in some sense, the emergent behavior of complex systems.

A good pattern language guides designers toward good system architectures: ones that are useful, durable, functional, and aesthetically pleasing.

Pattern languages are critical because they capture, in some sense, the emergent behavior of complex systems: "The combination of patterns acting on a smaller level of scale acquires new and unexpected properties not present in the constituent patterns."[17] A pattern language is thus a collective of solutions to recurring problems, each in a context and governed by forces. The solutions work together at every level of scale to resolve a complex problem. A good pattern language guides designers toward good system architectures: ones that are useful, durable, functional, and aesthetically pleasing.[18]

In other words, the whole is more than the sum of the parts: a pattern language is more than the sum of its patterns.

7. A pattern is validated by use

Patterns are usually discovered through concrete experience rather than abstract argument, although both are possible.[5] But a pattern cannot be verified or validated from a purely theoretical framework. In the end, such proof of a pattern's existence lies in its recurring, identifiable presence in artifacts.

In a spoken language, new words are devised, or old words acquire new meanings, through common use. New words can also be created "theoretically"—for example, by combining appropriate word roots—but any newly minted word is not validated unless and until it achieves widespread use. So, too, a pattern's repeated presence in existing artifacts confirms its usefulness. Theory is important, but to be meaningful it must be evaluated in the context of concrete experience.

Consider the technique for solving difficult problems that Floyd outlined in his 1978 Turing Lecture:

> *In my own experience of designing difficult algorithms, I find a certain technique most helpful in expanding my own capabilities. After solving a challenging problem, I solve it again from scratch, retracing only the insight of the earlier solution. I repeat this until the solution is as clear and direct as I can hope for. Then I look for a general rule for attacking similar problems, that would have led me to approach the given problem in the most efficient way the first time. Often, such a rule is of permanent value.*[8]

Floyd's technique offers insight into what it means for a pattern to be validated by use. At the point where he has first solved the challenging problem, Floyd might have discovered a pattern. But not until he has discovered a "general rule for attacking similar problems"[8] and used that rule in other situations can he call his solution a pattern.

Alexander argues that in confirming the existence of architectural patterns "we must rely on feelings more than intellect."[5] He is precise in what he means by this—there is no simple rule with which to verify a pattern's existence. Confirming the existence of architectural patterns is, therefore, not simply a process of abstract argument; it requires a more intangible mix of theory and practice.

Schmidt notes that "Patterns are validated by experience rather than by testing, in the traditional sense of 'unit testing' or 'integration testing' of software."[11] However, he also states that validating a pattern's existence by experience is difficult, because it is hard to know when a pattern is complete and correct. His group used periodical reviews of patterns to help with that process.

Each pattern in the *Design Patterns* catalog lists its known uses: examples of the pattern in real systems. This provides a critical check on the pattern's validity. In contrast, Ralph Johnson uses what he calls patterns to document object-oriented framework use, but gives no evidence that his patterns have occurred in more than one solution—that is, are used by a number of users of the HotDraw framework.[4] To validate his patterns, he should give concrete examples of where they have been used. Then, he will have shown that they are useful.

8. A pattern is grounded in a domain

A pattern is not an isolated entity. It is defined both in the context of other patterns (a pattern language) and with respect to a particular area or field to which it applies. Discussion of a pattern only makes sense as part of a pattern language. Moreover, discussion of a pattern has no meaning outside the domain to which it applies.

For example, *Design Patterns* describes patterns in the domain of object-oriented software construction, whereas Johnson describes a set of patterns in the domain of framework use. Johnson's HotDraw pat-

terns work well together, as do those from the *Design Patterns* catalog, but combining one from each makes no sense.

A discussion of patterns must clarify what domain the patterns serve, and it must ensure that all patterns share a common domain. Otherwise, the discussion will likely be confused and confusing.

9. A pattern captures a big idea

Design patterns are not about solutions to trivial problems, so not every solution to a software design problem warrants a pattern. Rather, patterns focus on key, difficult problems in a particular area—problems that designers in that area face time and time again, in one form or another. Thus, a pattern language "captures" a domain: together, the patterns in the language identify the domain's key concepts and the important aspects of their interplay.

The elements of other languages exhibit the same effect. For example, the key words of a spoken language—the nouns, verbs, and adjectives that carry much of the meaning in communication—correspond to key objects, actions, and descriptions that occur repeatedly. It is not necessary or even sensible to make up a new word for every concept; instead, we combine existing words in phrases and clauses that convey meaning.

Consider the problem, in an OO context, of extending an object's behavior. Often, the solution is simple—create a subclass with the extra behavior—and does not require a pattern. But this solution does not let existing objects take advantage of the extra behavior, because existing objects will inherit from the (nonextended) base class. If, instead, we add the extension to the base class itself, existing objects can use the new behavior, but the result can be an unwieldy base class that tries to be all things to all clients.

The more complex problem of extending an object's behavior without modifying the base class, such that existing objects that choose to do so can access the extended behavior, does warrant a pattern-based solution. It is a key problem in OO design and is addressed by Erich Gamma's Extension Object pattern.[19] This pattern lets an extension object's clients choose and access the interfaces they need by defining extension objects and their interfaces in separate classes.

A pattern must strike a balance. It must propose a specific solution to a specific problem. But if the problem it addresses is not significant, the pattern approach's impact is lost.

A pattern must strike a balance. It must propose a specific solution to a specific problem.

Developing a definition that completely captures the pattern concept is neither worthwhile nor possible. But it is possible and worthwhile to document essential characteristics of patterns as a means of clarifying and developing our understanding of the concept and thus our ability to identify and use patterns.

How do our characteristics help answer the questions posed in the introduction? In our view, Mediator exhibits all the characteristics—a pattern language for OO software design will likely include this pattern. So, too, we would likely find Bubblesort in a language of algorithm-like patterns and Pipe and Filter in a pattern language for software architecture. Extend by Subclassing, however, falls short because it does not capture a big idea—including it in a language for software design would dilute the impact of more important patterns. Johnson's HotDraw patterns fall short on two grounds. As we mentioned before, Johnson provides no evidence that they have been validated by repeated use, and they do not clearly identify the domain (framework use or design) in which they apply.

References

1. E. Gamma et al., "Design Patterns: Abstraction and Reuse of Object-Oriented Design," *Proc. 1993 European Conf. Object-Oriented Programming* (ECOOP 93), Lecture Notes in Computer Science, vol. 707, Springer-Verlag, Heidelberg, Germany, 1993, pp. 406–431.
2. K.J. Sullivan, I.J. Kalet, and D. Notkin, "Evaluating the Mediator Method: Prism as a Case Study," *IEEE Trans. Software Eng.*, vol. 22, no. 8, Aug. 1996, pp. 563–579.
3. M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, Upper Saddle River, N.J., 1996.
4. R. Johnson, "Documenting Frameworks Using Patterns," *Proc. 1992 Conf. Object-Oriented Programming: Systems, Languages, and Applications* (OOPSLA 92), *ACM Sigplan Notices*, vol. 27, no. 10, Oct. 1992, pp. 63–76.

ADVERTISER INDEX
JANUARY / FEBRUARY 2002

IEEE Software
IEEE Computer Society
10662 Los Vaqueros Circle
Los Alamitos, California 90720-1314
Phone: +1 714 821 8380
Fax: +1 714 821 4010
http://computer.org
advertising@computer.org

5. C. Alexander, *The Timeless Way of Building*, Oxford Univ. Press, New York, 1979.
6. J.O. Coplien, *Software Patterns*, SIGS Books & Multimedia, New York, 1996.
7. E. Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, Mass., 1995.
8. R.W. Floyd, "The Paradigms of Programming," *Comm. ACM*, vol. 22, no. 8, Aug. 1979, pp. 455–460.
9. S. Henninger, "Using Iterative Refinement to Find Reusable Software," *IEEE Software*, vol. 11, no. 5, Sept. 1994, pp. 48–59.
10. C. Rich and R.C. Waters, *The Programmer's Apprentice*, Addison-Wesley, Reading, Mass., 1990.
11. D.C. Schmidt, "Experience Using Design Patterns to Develop Reusable Object-Oriented Communication Software," *Comm. ACM*, vol. 38, no. 10, Oct. 1995, pp. 65–74.
12. C. Alexander, *Notes on the Synthesis of Form*, Harvard Univ. Press, Cambridge, Mass., 1964.
13. T. Winograd and F. Flores, *Understanding Computers and Cognition*, Ablex Publishing, Norwood, N.J., 1986.
14. W. Pree, *Design Patterns for Object-Oriented Software Development*, Addison-Wesley, Reading, Mass., 1995.
15. C. Alexander, S. Ishikawa, and M. Silverstein, *A Pattern Language*, Oxford Univ. Press, New York, 1977.
16. D. Lea and C. Alexander, "An Introduction for Object-Oriented Designers," *Software Eng. Notes*, vol. 19, no. 1, Jan. 1994, pp. 39–52.
17. N.A. Salingaros, "Structure of Pattern Languages," *Architectural Research Quarterly*, vol. 4, no. 2, 14 Sept. 2000, pp. 149–162.
18. B. Appleton, "Patterns and Software: Essential Concepts and Terminology," 2000, www.enteract.com/~bradapp/docs/patterns-intro.html (current Nov. 2001).
19. E. Gamma, "Extension Object," *Pattern Languages of Program Design 3*, R. Martin, D. Riehle, and F. Buschmann, eds., Addison-Wesley, Reading, Mass., 1998, pp. 79–88.

About the Authors

Tiffany Winn is a PhD student in computer science at Flinders University, where she received her BSc (Hons.) in computer science. Her research interests are in design patterns and programming paradigms. Contact her at the School of Informatics and Eng., Flinders Univ., GPO Box 2100, Adelaide SA 5001, Australia; winn@cs.flinders.edu.au; www.cs.flinders.edu.au/People/Tiffany_Winn.

Paul Calder is a senior lecturer in the School of Informatics & Engineering at Flinders University. His research interests include object-oriented software design, component-based software reuse, graphical interfaces, and data visualization. He is a member of the IEEE Computer Society, ACM SIGCHI, and ACM SIGPLAN. He earned his PhD in electrical engineering from Stanford University, where he was one of the developers of the InterViews user interface toolkit. Contact him at the School of Informatics and Eng., Flinders Univ., GPO Box 2100, Adelaide SA 5001, Australia; calder@cs.flinders.edu.au; www.cs.flinders.edu.au/People/Paul_Calder.