Software Pre-Patterns as Architectural Knowledge

Gerald Bortis and André van der Hoek
University of California, Irvine
Department of Informatics
Irvine, CA 92697-3440
{gbortis,andre}@ics.uci.edu

ABSTRACT
Christopher Alexander's introduction of patterns inspired their application in fields such as software engineering. However, their current realization deviates from his original intent in how and when they are used. In this paper, we contrast Alexander's concept of patterns to their current realization in software engineering and suggest a new approach to creating patterns which are broader and can be applied at the early phases of the design process, and thus adhere to Alexander's original intent as a format for capturing and sharing important design knowledge.

Categories and Subject Descriptors
D.3.3 [**Programming Languages**]: Language Constructs and Features – *Patterns*

General Terms
Design, Documentation

Keywords
Patterns, design patterns, pre-patterns, architectural knowledge

1. INTRODUCTION
Patterns have become a powerful and ubiquitous concept in software engineering. The notion of a *pattern*, or "a rule which expresses a relation between a certain context, a problem, and a solution" was first introduced in the field of architecture by Christopher Alexander in his seminal book *The Timeless Way of Building* with the goal of providing an approach to designing buildings and towns which posses an intrinsic quality which can not be named, in other words, structures which are "alive." In his work, Alexander establishes a formal framework for capturing and defining these patterns so that they can be shared and provide insights into design problems [1].

The software engineering community at large has since readily adopted patterns. The most notable of these efforts has been the introduction of software design patterns described in *Design Patterns* [8]. Patterns have also emerged in other areas such as user interface design [4], web design [9], and ubiquitous computing [6]. In each of these areas patterns provide a structured format for capturing and sharing design knowledge between practitioners.

As the adoption of patterns continues to spread through the software engineering community, it is important to understand how they are being used, and how they deviate from the original intent of Alexander's approach. In this paper, we contrast Alexander's use of patterns in architecture with their current realization in software engineering, and suggest a new approach to creating and using patterns that is truer to their original intent.

2. TIMELESS PATTERNS
Alexander describes patterns as common recurring elements that contribute to endowing a building or town with an unnamable quality that is difficult to describe but trivial to detect. These patterns are intertwined in a *pattern language* that composes patterns and facilitates the design of complex "living" structures through the exploration of a web of dependent patterns. Architects can communicate their knowledge through this language that, over time, is envisioned to become common to all designers. A pattern is considered stable when it allows its own internal forces to resolve themselves. For example, a window seat addresses the potentially conflicting forces involving a person's natural tendency toward light and the desire to sit and be comfortable. A structure can only be "alive" when all of its composing patterns are stable. Since these qualities are ones which are also desired in software systems, patterns have proven useful in expressing essential knowledge of recurring problems in software engineering. However, current applications of patterns differ from Alexander's in several ways.

The first important difference is the timing of patterns, or the phases of the design process at which a designer can employ patterns. Alexander describes patterns as "rules of thumb" which exist in the designer's mind in the form of an ideal structure [2]. In describing his own process of designing a small cottage, he begins by first listing the patterns "in the order they come in, one at a time" given the requirements of the scenario, such as the state of the existing structure, space available, cost of materials, etc. In this early phase of design, he chooses a set of patterns he wishes to utilize in his design, such as THE FAMILY and NUMBER OF STORIES. Here we see that patterns are employed in the earliest phase of the design process when the requirements are being identified. He even introduces the SITE REPAIR pattern that solves the problem of preparing the site for the new structure. Such patterns do not provide guidance in constructing the cottage, but rather in preparing a design for the cottage, and span the entire life of a design, from the designers mind to the last detail.

Alexander's use of patterns in these examples differs significantly from current approaches to applying patterns in software engineering, which tend to focus heavily on object-oriented models of software, and the creation of patterns that address implementa-

SHARK'08, May 13, 2008, Leipzig, Germany.

tions based on these models. Patterns in software engineering are often expressed through code fragments and accompanied by UML diagrams that address the implementation. This approach to patterns prohibits the designer from considering patterns during the iterative process of gathering requirements and forming an early architecture because the patterns address problems and make use of concepts and terms that are encountered during implementation. Design patterns such as SINGLETON and FACTORY are conceptually near to code, and fail to address problems encountered in the early phases of design that involve making important high-level decisions.

The second important characteristic of Alexander's patterns is their level of detail. Alexander describes the use of patterns as the process of "differentiating space" where very high-level patterns are applied to an open space. As the space becomes more defined, lower-level patterns can be applied. This process is accomplished using a pattern language consisting of high-level patterns that are composed of dependent lower-level patterns. For example, in designing a structure, windows cannot be placed before a wall has been designed, and a wall cannot be designed before the axis of the building has been decided. Alexander writes, "to make a design, you take the patterns one by one, and use each one to differentiate the product of the previous patterns." We notice that such patterns support every level of the decision making process.

The current realization of patterns in software engineering should be viewed as low-level patterns that require high-level decisions to be made before hand. Rather than spanning every level of the decision making process, they require an architecture, and for the space to be already differentiated, thus causing a discontinuity in the design process.

From these two observations we note that Alexander's patterns are more effective, and actually aimed at, sharing insights to a solution to a design problem, rather than structuring the implementation of the problem, which is how they have been realized in software engineering. This difference, however subtle, is important because it addresses the nature of a pattern and its ultimate goal. To effectively foster the sharing of design knowledge and support the growth of a pattern language, patterns must address these issues in any field.

3. RELATED WORK

There have been attempts to adopt patterns at a higher level and earlier in the design process. For example, there are efforts to identify, name, and analyze patterns in software architectures as architectural styles or idioms [10]. Architectural styles are presented as an abstraction of elements from various specific architectures and are less constrained and less complete than a specific architecture [5]. Even though architectural styles strive to encapsulate high-level decisions and constraints of the elements, they still differ from Alexander's patterns in that they are often singular in nature. For example, choosing a style such as CLIENT-SERVER or BLACKBOARD as part of the design process is the limited extent of an architectural style. Alexander's patterns are hierarchical, with high-level patterns being composed of more detailed patterns, and they are applied throughout the design process.

Domain-specific software architectures, or DSSAs, are another example of abstractions meant to address similar issues. DSSAs provide an assemblage of software components, specialized for a particular domain in a standardized structure for building applications, as well as "a context for patterns of problem elements, solutions elements, and situations that define mappings between them" [11]. While DSSAs are useful in providing a common language that can be used to share knowledge of a particular domain, they are at the same time constrained since they focused on a single domain. In contrast, pattern languages are to be applied across any domain. For example, the same patterns could be used for a house as for a church. Furthermore, DSSAs require that the domain be thoroughly modeled so that reference architectures can be created. This deviates from Alexander's original vision of patterns as genetic code that guides the growth of a structure, rather than blueprints that determine the final plan [2].

The goal of design rationale is to capture the knowledge and reasoning justifying a resulting design, such as how a design met quality requirements and why certain designs were selected over others [3]. While design rationale, once collected, can be used similar to and even expressed as pre-patterns, they differ in that they are constructed after the fact. Pre-patterns are instead meant to guide the decision making process as the designer selects various patterns to apply to the problem at hand.

Finally, pre-patterns share characteristics with the concept of analysis patterns. Both pre-patterns and analysis patterns are intended to be applied across domains and help the designer to better understand the problem by capturing the conceptual structures of the business process rather than actual software implementations [7]. However, analysis patterns are heavily oriented towards the modeling of objects and their relationships, and do not directly address other concerns such as architectures or user interfaces. Pre-patterns have broader applicability and are meant to be less precise regarding the models which are used in a design.

4. PRE-PATTERNS

Similar approaches to applying patterns have been made in other areas of computer science as well. In attempting to identify common design problems that pertain to ubiquitous computing, Eric et al. have developed the concept of *pre-patterns* to refer to "structured design knowledge in a nascent domain" [6]. By surveying products related to ubiquitous computing, the authors identified 48 common designs that follow Alexander's structure of describing a particular design problem, as well as the forces that act upon it, and a solution. They address problems particular to the field of ubiquitous computing, such as DURABILITY AND ROBUSTNESS and GROUP CALENDAR. The pre-patterns also tend to "focus on high-level issues, such as user needs, versus specific user interfaces and interaction techniques," because "many of these high-level issues are better understood than the low-level techniques for implementing them" [6]. An empirical study of the effectiveness of these pre-patterns has demonstrated that they aid novice designers unfamiliar with an application domain to communicate ideas easier and to avoid some common problems in their design. In particular, the patterns prove to be effective in the idea generation process by assisting in solving high-level design problems [6].

This notion of a pre-pattern that provides guidance during the design process is a step in the right direction towards applying Alexander's original patterns in software engineering. In this paper we examine the potential of pre-patterns in software engineering, particularly as related to the problem of sharing architectural knowledge. Pre-patterns are meant to supplement existing patterns in software engineering since they address the problems which occur early in the design process, and at a higher-level than existing design patterns. They achieve this by providing multiple solu-

tions, providing a higher level of abstraction, and by addressing the design problem rather than the implementation.

Multiple solutions. Pre-patterns address one problem with multiple potential solutions. Design patterns provide a single solution to a problem, leaving only implementation details up to the designer. This is not flexible and does not support the creative design process that involves the designer evaluating multiple solutions. Since pre-patterns are used early in the design process, they support creativity by presenting multiple solutions to a problem.

Level of abstraction. Pre-patterns provide a level of abstraction that is higher than design patterns. Whereas design patterns provide code fragments and UML diagrams, pre-patterns do not need to acknowledge model or paradigm specific concepts like objects and interfaces that are encountered during implementation. At the same time, pre-patterns can be contained within architectural styles, which themselves can be represented as pre-patterns.

Problem solution. Because pre-patterns are used early in the design process and at a higher level, they are inherently more useful in structuring a solution given the context, rather than addressing the problems related to the implementation. They are conceptually closer to requirements than design patterns and can be used to convey the problem using a pattern language more accurately, which is a closer realization to Alexander's original concept of patterns that address the forces that act upon a context.

5. EXAMPLES

To illustrate these features, we provide several examples of pre-patterns that have been observed by one of the authors in the design and implementation of Mirth, an open source healthcare integration engine. Mirth has been in continuous design and development since early 2006 and is used in production in numerous hospitals and health care organizations.

We present these pre-patterns in a format similar to that in *A Pattern Language* because it conveys the intent of patterns by describing: the context or essence of the problem, examples for validity, and a set of potential solutions which describes the relationships required to solve the stated problem in the stated context [1].

CONCURRENT MODIFICATION

Two or more people using an application simultaneously will need access to the same data.

Web-based applications often allow multiple people to access the content. For example, an order tracking system might allow the salesperson and the customer to view and modify the same order simultaneously. The ability to simultaneously use an application suggests that concurrent modification of underlying data can occur.

Therefore, this problem may be addressed in several ways (note: there are additional solutions beyond the three listed here; for brevity we omit those at this time):

- locking the data and allowing only one user to view or modify it at any time,
- allowing each user to store a local copy of the data as they view and modify it, and merging the modified version with the shared version at a later time,
- providing all users with access to the data and ignoring any concurrency issues.

LARGE QUERY RESULTS

Searches may return a large number of results that take a long time to transmit between systems or to display.

Searching for and viewing items is a common feature of many applications. In some situations, the searches may return a large number of results. For example, using a library access system to view all books on “software engineering” would return a significant number of results that cannot all be retrieved or displayed.

Therefore, this problem may be addressed in several ways:

- paging (splitting the results into pages of a set size) the results of the query either locally on the system which is viewing the results, or remotely on the system which is providing the results,
- limiting the number of results returned.

ROLES AND PRIVILEGES

People need different levels of access to different parts of the system.

For security or privacy reasons, different people may not be allowed to access specific parts of a system. For example, in an employee timesheet system, only the manager should be able to view all of the employee’s hours. The employees should only be able to enter their own hours.

Therefore, this problem may be addressed by creating roles with assigned privileges. For example, only the manager will be assigned the role that is given the privilege to access the employee overview page.

LARGE DATA STREAMS

An application may receive a large data stream for processing.

Event-driven applications that receive data from external sources may be subject to overwhelming amounts of incoming data. For example, a message routing application may receive more messages than it can store or process.

Therefore, this problem may be addressed in several ways:

- by keeping a buffer of a preset size to store data when its size has exceeded processing limitations,
- by rejecting any new incoming data, optionally with a notification to the sending system.

In elaborating these pre-patterns, we notice several key features. First, they are broader than design patterns in their context. Pre-patterns do not assume a specific programming paradigm or implementation model. Instead, they provide insights which are useful in guiding the decision making process and in forming a solution. For example, the LARGE QUERY RESULTS pre-pattern represents a characteristic that is exhibited by numerous distributed systems. By having this pre-pattern present during the thinking process one can quickly examine the problem at hand in his or her head for the potential presence of this issue. With a checklist of pre-patterns, it then becomes possible to avoid many surprises later in the development lifecycle when the actual architecture and design are much more complete and would possibly need to change much in order to accommodate this kind of concern.

Second, several of the pre-patterns suggest several possible solutions. These solutions are intended to stimulate the designer into considering possible alternatives, and in creating his own. We also

note the composability of these pre-patterns. For example, CONCURRENT MODIFICATION should be considered along with LARGE QUERY RESULTS in situations where multiple users may edit the results returned from the search. Here we see that applying pre-patterns to a design problem is a process similar to Alexander's "differentiating space." Unlike approaches that attempt to differentiate the space only once during the design process, pre-patterns can be combined in many different ways as new design problems appear. For example, Mirth was built using an overall architectural style, CLIENT-SERVER, but pre-patterns still naturally fit and in many ways inspired the choice of style and subsequent refinement and adjustment of the Mirth design as problems such as LARGE QUERY RESULTS due to message volume and CONCURRENT MODIFICATION due to threading became apparent.

6. DISCUSSION

Based on these observations, we propose that pre-patterns should be explored as a viable tool for sharing architectural knowledge, since the application of pre-patterns automatically conveys knowledge pertaining to why a particular design was chosen in solving a design problem. This knowledge of the solution, rather than the implementation, is immensely valuable in better understanding the decisions that are made when architecting a software system.

Like Alexander's original patterns, pre-patterns provide a standard design vocabulary that can be shared among designers. As practitioners apply pre-patterns, they can recognize and create new pre-patterns that further convey knowledge of architectural design problems. This process of applying and reformulating patterns, or what Alexander calls the "evolution of a common language," is the key insight into effectively sharing architectural knowledge. Pre-patterns help bridge the gap between the designer and the user by providing representations of the solution that are discussable and open to criticism.

Finally, as stated before, because pre-patterns are applied during the early phases of design, they can also be used to make the designer aware of problems that he may not have been considering in developing the solution. This unique state in which pre-patterns exist, distinct from design patterns and architectural styles, allows them to act as an awareness mechanism for architectural knowledge.

7. CONCLUSIONS AND FUTURE WORK

In this paper we introduced the concept of pre-patterns for software engineering. We believe pre-patterns are a closer realization of Alexander's original approach to conveying important design knowledge through patterns that can be applied at the earliest phases of design. Pre-patterns also provide the designer with high-level abstractions that are oriented towards framing the problem rather than the implementation. When applied, their broad applicability also encourages the evaluation of multiple potential solutions to a design problem. These distinguishing attributes make pre-patterns a useful vehicle for expressing and sharing architectural knowledge.

We realize this introduction is only the beginning of our work. To evolve the concept of pre-patterns, we want to continue to collect new pre-patterns based on our involvement in the design of software systems such as Mirth. Specifically, we want to reconstruct Mirth completely and in detail in terms of pre-patterns at various levels of abstraction, and do so as well for other systems to understand which pre-patterns play a role when and where. With this kind of knowledge we can better address the ultimate role we envision for pre-patterns: how they can enhance the design process, particularly early on but as much so in their effect on later phases of the design process.

Because pre-patterns are broad in scope and are applied during the early phases of the design process, we realize their potential in further avenues of research. We wish to further analyze the value of pre-patterns in representing requirements or concerns during design and in enabling stakeholders to better understand and track them before they are further formalized. We also wish to investigate various methods of applying pre-patterns during design, including potential support through tools that expose designers to common problems and to foster creativity, with the ultimate goal of supporting the design of higher quality software applications.

8. ACKNOWLEDGEMENTS

Effort partially funded by the National Science Foundation under grant numbers DUE-0536203 and IIS-0534775.

9. REFERENCES

[1] Alexander, C., *A Pattern Language: Towns, Buildings, Construction*. 1977: Oxford University Press.

[2] Alexander, C., *The Timeless Way of Building*. 1979: Oxford University Press.

[3] Antony, T., et al., *A survey of architecture design rationale*. J. Syst. Softw., 2006. 79(12): p. 1792-1804.

[4] Borchers, J., *A Pattern Approach to Interaction Design*. 2001: John Wiley and Sons.

[5] Dewayne, E.P. and L.W. Alexander, *Foundations for the study of software architecture*. SIGSOFT Softw. Eng. Notes, 1992. 17(4): p. 40-52.

[6] Eric, S.C., et al., *Development and evaluation of emerging design patterns for ubiquitous computing*, in *Proceedings of the 5th conference on Designing interactive systems: processes, practices, methods, and techniques*. 2004, ACM: Cambridge, MA, USA.

[7] Fowler, M., *Analysis Patterns: Reusable Object Models*. 1997: Addison-Wesley.

[8] Gamma, E., Helm R., Johnson R., and Vlissides J., *Design Patterns*. 1995: Addison-Wesley.

[9] Graham, I., *A Pattern Language for Web Usability*. 2003: Addison-Wesley.

[10] Gregory, D.A., A. Robert, and G. David, *Formalizing style to understand descriptions of software architecture*. ACM Trans. Softw. Eng. Methodol., 1995. 4(4): p. 319-364.

[11] Will, T., *DSSA (Domain-Specific Software Architecture): pedagogical example*. SIGSOFT Softw. Eng. Notes, 1995. 20(3): p. 49-62.