CHAPTER

1

The Pattern Approach

It is not necessarily complicated. It is not necessarily simple.

Christopher Alexander, in 'The Timeless Way of Building'

In this chapter we introduce the concepts of patterns and two approaches to organizing and connecting them: pattern systems and pattern languages. In addition, we outline the major application areas and purpose of patterns, as well as their history in the software community. Last, but not least, we discuss how patterns are mined, documented, and prepared for publication and presentation.

1.1 Patterns at a Glance

Developer enthusiasm for patterns has been almost unquenchable since the release of the seminal work by the Gang-of-Four[1] [GoF95] just a decade ago. Software developers from around the world leapt on the 'new idea,' with the hope that patterns would help them untangle tricky problems into a well-knit solution—something with elegance, directness, and versatility. Patterns found their way into many software development projects. A movement had begun. It was, and still is, thriving.

A major reason for the success of patterns is that they constitute a 'grass roots' initiative to build on, and draw from, the collective experience of skilled designers. It is not often that a new development project tackles genuinely new problems that demand truly novel solutions. Developers may sometimes arrive at similar solutions independently or often recall a similar problem they solved successfully in a different situation, reusing its essence and adapting its details to resolve the new problem. Expert developers can draw on a large body of such solution schemes for both common and uncommon design problems. This practical experience guides them when building new applications.

Distilling commonalities from the pairing of application-specific design problems and their solutions leads comfortably to the concept of patterns: they capture these solutions and their relationship to the problem, framing them in a more readily-accessible form. From a very general birds-eye perspective, a pattern can be characterized as:

A solution to a problem that arises within a specific context.

Though this characterization captures every pattern's main structural property well, it does not tell the whole story. The context-problem-solution trichotomy is necessary for a specific concept to qualify as a pattern, but it is not sufficient. In particular, it does not specify how to distinguish a true pattern from an 'ordinary' solution to a problem. In fact, it requires much more for a software concept to be a true pattern:

- A pattern describes both a process and a thing: the 'thing' is created by the 'process' [Ale79]. For most software patterns—thus also for security patterns—'thing' means a particular high-level design outline or code detail, including both static structure and intended behavior. In other words, a pattern is both a spatial configuration of elements that resolve a particular problem—or in which a particular problem does not arise—and a set of associated instructions to create this configuration of elements most effectively.

[1] The authors of this book, Erich Gamma, Ralph Johnson, Richard Helm, and John Vlissides, are named after the 'Gang-of-Four' in Chinese politics.

- A true pattern presents a high-quality, proven solution that resolves the given problem optimally. Patterns do not represent neat ideas that might work, but concepts that have been applied successfully in the past over and over again. Consequently, new ideas must first prove their worth in the line of active duty, often many times, before they can truly be called patterns. Because they capture practice and experience, patterns can help novices to act with greater confidence and insight on modest-sized projects, as well as supporting experts in the development of large-scale and complex software systems.
- Patterns support the understanding of problems and their solutions. Presenting a problem and a solution for it is not enough for a pattern, as this leaves several important questions unanswered. Why is the problem a hard problem? What are the requirements, constraints, and desired properties of its solution? Why is the solution as it is and not something else? A good pattern does not withhold this information. The forces associated with its problem description provide the answer for the first two questions, and the discussion, or consequences, of its solution the latter.
- Patterns are generic—as independent of or dependent on a particular implementation technology as they need to be. A pattern does not describe a particular solution, a specific arrangement of components or classes dependent on a particular programming paradigm or language, but a set of interacting roles that define an entire solution space. Christopher Alexander puts it this way [AIS+77]: 'Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it twice the same.'
- A pattern tells a story and initiates a dialog. As every pattern presents timeless and proven experience, it tells a success story. To be precise for software patterns, a 'successful software engineering story,' to borrow an observation from Erich Gamma. But a pattern is not only a story, it also initiates a dialog with its readers about how to resolve a particular problem well—by addressing the forces that can influence the problem's solution, by describing different feasible solutions, and finally by discussing the trade-offs of each solution option. A pattern thus invites its readers to reflect on the problem being presented: to think first and then to decide and act explicitly and consciously.
- Patterns celebrate human intelligence. Patterns are not automatic derivations from problem ingredients to fully-baked solutions. Patterns often tackle problems in more lateral ways that can be indirect, unusual, and even counter-intuitive. In contrast to the implied handle-turning nature of many rigid development methods, patterns are founded in human ingenuity and experience.

A true pattern exposes all of the above properties—if it is lacking any of them, it is probably just a solution to a problem, and most likely a specific design and implementation decision for a specific system, but not a pattern. Adapting the existing definition from the first volume of the *Pattern-Oriented Software Architecture* series [POSA1], this leads to the following characterization of the notion of patterns:

A pattern for software architecture describes a particular recurring design problem that arises in specific design contexts, and presents a well-proven generic solution for it. The solution consists of a set of interacting roles that can be arranged to form multiple concrete design structures, as well as a process for creating any particular structure.

This general definition serves well for the purpose of this book, although we narrow it to security patterns but also extend it to include enterprise and requirements patterns as well as architecture.

1.2 No Pattern is an Island

Though each pattern focuses on providing a self-contained solution for resolving one specific problem, patterns are not independent of one another. In fact, there are many relationships between patterns [POSA1]. The most important relationship is refinement: the solution proposed by a particular pattern can often be implemented with help of other patterns, which resolve sub-problems of the original problem. To put it in another way, 'each pattern depends on the smaller patterns it contains and on the larger patterns in which it is contained' [Ale79]. Other important relationships among patterns are variation and combination [POSA1].

It is the relationships between the patterns, together with their genericity, that allows them to be combined and integrated with one another to form large software architectures and designs that are coherent and consistent in their whole as well as in their details. Conversely, without these relationships, patterns would only be able to resolve isolated problems, with no, or at best limited, effect on a larger design or even an entire software architecture [POSA4].

1.3 Patterns Everywhere

Software patterns can exist at any scale and for many problem areas. In their early days—the mid 1990s—the focus was on object-oriented design patterns of general applicability. The Gang-of-Four book [GoF95] presents the most widely-known patterns of this kind. The scope of these patterns, however, had only a small impact on

entire software or system architectures, because they covered how to structure specific components, or how to organize their relationships and interactions. This gap was first filled by the POSA team [POSA1] who were the first to present patterns at the level of coarse-grained software architecture. Other authors completed the pattern space to the 'bottom,' the level of programming. For example, James O. Coplien—commonly known as 'Cope'—published patterns that deal with C++-specific issues like memory management and string handling [Cop92], and Kent Beck wrote his famous patterns book *Smalltalk Best Practices* [Bec97]. Yet all these patterns, although covering different scopes, were general in nature: general architecture, general design, and general programming.

The first patterns that had a more specific focus were Martin Fowler's analysis patterns [Fow97]. Not only did Martin introduce another level of scope to patterns—patterns that describe the fundamental structure of, and workflow within, an application domain—his patterns focused on two specific areas: health care and corporate finance.

Since then patterns have spread into many other specific areas, ranging from concurrent and networked systems and programming [Lea99] [POSA2], server components [VSW02], human–computer interaction [Bor01], memory constraint systems [NW01], resource management [POSA3], and others [Ris00]. Recently, security was identified as another area of hot interest for patterns, and this book is intended to serve this need.

1.4 Humans are the Target

A valid question to ask is 'What is the target audience for patterns.' There are many answers to this question, depending on who you ask, but all leading pattern experts share a common view: patterns are for, and about, humans. This statement is also consistent with Christopher Alexander's—the architect, in the sense of buildings, not software, who invented patterns—understanding of patterns [Ale79].

The correctness of this position becomes obvious when reflecting on the previous sections of this chapter. All the properties of patterns that we have discussed so far are aimed at presenting problems and their solutions in a way that humans can understand: when such problems arise, what the problems are, what to consider when resolving them, how they can be resolved, how their solutions are implemented, and why these solutions are as they are. Much effort is also expended in presenting patterns in an appealing, dialog-initiating, and story-telling—in other words, human-readable—form.

Humans are also the only audience for patterns. We discovered that it is next to impossible to formalize patterns, a necessary precondition to making patterns machine-readable and automatable. Thus the audience for patterns does not include computers or any other type of machine, nor the many software development tools that run on such computers or machines.

This distinguishes patterns from other design or modeling techniques, for example, the Unified Modeling Language [BRJ98]. Artifacts that are created with such techniques are not only intended to be human-usable and human-readable: they can also be input to tools that then execute formal consistency and correctness checks on them, simulate them, and even generate code fragments from them. At first glance such techniques might seem superior to patterns. However, in real-world practice they are only useful for documenting, implementing, and tuning an already-designed system. They do not support us in the creative act of designing a new system and understanding its challenges—but patterns do!

1.5 Patterns Resolve Problems and Shape Environments

Now that we know that software patterns intend to support humans in understanding and building software systems, we can ask what concrete purpose they serve in that context.

The most obvious—and of course correct—answer is: software patterns help humans to understand and resolve problems. Why else do they contain human-readable descriptions of problems and their solutions? The problem areas that software patterns address are the organizational, analysis, architecture, design, and programming aspects of software development.

However, software patterns do not just specify arbitrary solutions to software development problems. As we discussed in earlier sections of this chapter, a pattern represents proven and practiced experience—timeless solutions to recurring problems that can be implemented in many different ways—presented so that people can understand, and talk about, the problems, the solutions, and their influencing forces and trade-offs.

When analyzing the way in which software patterns resolve the problems they address, we see that they do this by shaping environments: patterns introduce spatial configurations of elements that exhibit specific behavior. From a system development perspective we can also say: when applied, a pattern transforms a given structure in which a particular problem is present into another structure in which this problem is resolved.

Some pattern experts take this observation as an argument to invert the perspective, to better emphasize the focus on humans that patterns have: patterns shape environments in which particular problems do not occur, and in which humans thus feel comfortable.

Which perspective best serves you, or the particular application under development or refactoring, is a matter of your own preference. If you reflect on them long enough, however, you will discover that they are mutually supportive. With patterns, developers are more confident of avoiding problems or resolving them well, while customers and users are more confident that problems are avoided or resolved well. Thus both camps feel more comfortable that they are getting the ‘right’ thing.

1.6 Towards Pattern Languages

Experience in developing software with patterns reveals that the explicit relationships that can exist between patterns, as outlined in Section 1.2, *No Pattern is an Island*, are not enough to use patterns successfully. The reason for this is the existence of additional implicit relationships between patterns. When developing a real-world system, not only one design and implementation problem must be resolved, but many different and orthogonal ones. If we resolve one problem by applying a pattern and implementing it in a specific way, this creates a concrete design. This design then defines a framework for resolving subsequent problems—which, unfortunately, narrows their potential solution space. Consequently, it can happen that it is impossible to resolve the subsequent problems most optimally—or not even good enough—due to the constraints set by the existing design.

The patterns community tried to address this fact by structuring and organizing the pattern space. The goal of all such activities was to achieve a better overview of the patterns that exist for resolving a particular problem, and to elaborate how patterns can be combined into meaningful larger structures. Pattern catalogs [GoF95] and pattern systems [POSA1], therefore, present more than one pattern for resolving important design problems, for example object creation or location-independent inter-process communication. Pattern systems also discuss how to best combine their constituent patterns to form concrete software architectures and designs. The following is the original definition for pattern systems [POSA1]:

A pattern system for software architecture is a collection of patterns for software architecture, together with guidelines for their implementation, combination and practical use in software development.

Without delving into details, it is obvious from this definition that a pattern system can give a great deal more support for using patterns in practical software development than individual patterns can ever do. Yet pattern systems still do not address all the needs of a professional and holistic software development using patterns. In particular:

- What are the important design and implementation problems that arise during the development of a specific type of software system?
- How do all these problems relate to one another and in what order are they resolved most optimally?
- What (alternative) patterns can help to resolve each problem most effectively in the presence of the other problems?
- What are the criteria for deciding which of the alternative patterns for resolving a particular problem is most suitable in a given situation?

- How is the selected pattern instantiated most effectively within the existing (partial) software architecture?

Recognizing this leads almost directly to the concept of pattern languages [POSA4]. In a nutshell:

A pattern language is a network of tightly-interwoven patterns that defines a process for resolving a set of related, interdependent software development problems systematically.

For example, there are pattern languages that support

- Constructing entire applications, for example distributed systems [POSA4], or business information systems [Fow97]
- Developing major application parts such as components [VSW02]
- Addressing problem areas of common interest, such as security or human computer interaction [Bor01]
- Programming in a good style, for example, the Smalltalk best practice patterns [Bec97]

Basically, a pattern language exposes the same properties as an individual pattern, but at a higher, system-oriented level. It defines both a process and a thing, produces designs of high quality, allows the creation—or generation—of many alternative designs, supports the understanding of the challenges associated with a specific problem domain or the development of a specific application type, and tackles problems in an intelligent, often unusual and lateral way.

The following excerpt from [POSA4] summarizes the concrete look-and-feel of a high-quality pattern language.

> One or more patterns define the 'entry point' of the pattern language and address the most fundamental problems that must be resolved when building its 'thing'. The entry point patterns also define the starting point for the language's process: every software development that uses the language begins there. The creation process for the chosen entry point pattern then describes what concrete activities must be performed to resolve the specific problem that this pattern addresses. This process not only specifies how to implement the proposed problem resolution, however. It also suggests other patterns from the language that could be used to address sub-problems of the original problem, as well as the order in which to apply these other patterns. If several alternative patterns are referenced for resolving a particular sub-problem, the trade-offs of each alternative are described and hints are given for how to select the 'right' alternative for a specific application.

> Following any of the pattern suggestions made by the entry point pattern's creation process, the process defined by the pattern language leads to another pattern and its associated creation process—which is then applied to resolve the problem that the other pattern addresses. This process can reference yet more patterns, to resolve sub-problems of the sub-problem of the initial problem. Using either a breadth-first or a depth-first approach, or even a mixture of both approaches, this iterative process of following a pattern reference and applying the referenced pattern's creation process continues until there are no more pattern references to follow. The particular path taken through the pattern language then defines a sequence—or 'sentence'—of patterns that guides the design and implementation of the 'thing' that is this language's subject.

A pattern language is the highest organizational form for patterns currently known. It also the most successful organizational form with respect to the original goal of software patterns: to support the construction of real-world, productive software most professionally. For this reason, the patterns in this book are organized whenever possible into a pattern language, instead of just cataloguing them separately.

1.7 Documenting Patterns

Patterns, whether they are stand-alone or integrated into a pattern system or pattern language, must be presented in an appropriate form if we are to understand and discuss them. A good description helps us grasp the essence of a pattern immediately—what is the problem the pattern addresses, and what is the proposed solution? A good description also provides us with all the details necessary to implement a pattern, and to consider the consequences of its application.

Many pattern forms are known and used [POSA4], but for this book we decided to follow the format developed for the *Pattern-Oriented Software Architecture* series [POSA1], as it best fits our goals and audience:

Name

The name and a short summary of the pattern.

Also Known As

Other names for the pattern, if any are known.

Example

A real-world example demonstrating the existence of the problem and the need for the pattern. Throughout the description we refer to examples to illustrate solutions and implementation aspects, where this is necessary or useful.

Context

The situations in which the pattern may apply.

Problem

The problem the pattern addresses, including a discussion of its associated forces.

Solution

The fundamental solution principle underlying the pattern.

Structure

A detailed specification of the structural aspects of the pattern, using appropriate notations.

Dynamics

Typical scenarios describing the run-time behavior of the pattern.

Implementation

Guidelines for implementing the pattern. These are only a suggestion, not an immutable rule. You should adapt the implementation to meet your needs, by adding different, extra, or more detailed steps, or by re-ordering the steps. Whenever applicable we give UML fragments to illustrate a possible implementation, often describing details of the example problem.

Example Resolved

Discussion of any important aspects for resolving the example that are not yet covered in the *Solution*, *Structure*, *Dynamics*, and *Implementation* sections.

Variants

A brief description of variants or specializations of a pattern.

Known Uses

Examples of the use of the pattern, taken from existing systems.

Consequences

The benefits the pattern provides, and any potential liabilities.

See Also

References to patterns that solve similar problems, and to patterns that help us refine the pattern we are describing.

It is important to note that not all fields of this pattern form are mandatory. For example, not all patterns have alternative names or variants. Alternatively, for some patterns it is hard, or unnecessary, to provide detailed descriptions of its structure, behavior, and implementation, because all information can be integrated well into the core solution description. Likewise, if an example is embedded within every section of the form, there may not be a need for separate example sections.

Writing patterns is hard. Achieving a crisp pattern description takes several review and revision cycles. Many experts from all over the world have helped us with this activity, and we owe them our special thanks. Thus, we give credit to all who helped to shape a particular pattern in the introduction to each chapter.

1.8 A Brief Note on The History of Patterns

The architect Christopher Alexander laid the foundations on which many of today's pattern approaches are built. He, and members of the Center for Environmental Structure in Berkeley, California, spent more than twenty years developing an approach to architecture that used patterns. This 'entirely new attitude in architecture and planning' is published in a series of books [ANA+87] [AIS+77] [Ale79] [ASA+75]. Alexander describes over two hundred and fifty patterns that span a wide range of scale and abstraction, from structuring towns and regions down to paving paths and decorating individual rooms. He also defined the fundamental Context-Problem-Solution structure for describing patterns, the 'Alexander form.' Recently, some software pattern writers have started to distance themselves a little from Alexander, since they feel that his views on patterns do not translate directly into software patterns. They acknowledge the importance of Alexander's work, but would like to go their own way. Despite this discussion, however, Alexander's work is well worth reading by everybody who is interested in patterns.

The pioneers of patterns in software development are Ward Cunningham Kent Beck. They read Alexander's books and were inspired to adapt his ideas for software development. Ward and Kent's first five patterns deal with the design of user interfaces—their patterns WINDOW PER TASK, FEW PANES, STANDARD PANES, NOUNS AND VERBS, and SHORT MENUS mark the birth of patterns in software engineering.

Four software design experts—known as the 'Gang-of-Four' in the pattern community—paved the way for the wide acceptance of patterns in software engineering. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides are the authors of the seminal work *Design Patterns – Elements of Reusable Object-Oriented Software* [GoF95]. Many other software experts followed the path paved by the Gang-of-Four, as we briefly summarized in Section 1.3, *Patterns Everywhere*, producing an almost endless list of publications on patterns. At the current end of this path is this book. Its goal is to fill a still-blank spot on the world map of patterns: security.

1.9 The Pattern Community and its Culture

Software engineers all over the world are currently documenting their experience using patterns. This community is very active, interactive and supportive, with the goal of sharing and integrating knowledge, and spreading the word about successful software development practice.

The major forum of the pattern community is the PLoP (Pattern Languages of Programming) conference series, which is held in the US (the original PLoP and Chili-PLoP), Germany (EuroPLoP), Scandinavia (Viking PLoP), Brazil (Sugar Loaf PLoP), Japan (Mensore PLoP), and Australia (Koala PLoP). Its proceedings are published as a series of books [PLoPD1] [PLoPD2] [PLoPD3] [PLoPD4].

The culture celebrated by the pattern community—and consequently the culture of its PLoP conferences—differs significantly from other, more traditional cultures for presenting and discussing scientific work. It exhibits the following characteristics:

- A focus on practicability. The community looks for pattern descriptions of proven solutions to problems, rather than on presenting the latest scientific results.
- An aggressive disregard of originality. Pattern authors do not need to be the original developers of the solutions they describe.
- Non-anonymous review. Patterns are 'shepherded' rather than reviewed. The 'shepherd' contacts the authors of pattern papers and discusses the patterns with them. The goal is to improve the pattern such that it can be accepted for review at a PLoP conference and suffer as little rejection as possible.
- Writer's workshops instead of presentations. At PLoP conferences, patterns are discussed in writer's workshops made up of conference attendees, rather than being presented by their authors in open forum.
- Careful editing. Authors get the chance to include the feedback from the writer's workshops, and all patterns are copy-edited before they appear in the final conference proceedings or other book publications.

Most patterns presented in this book were discussed at the PLoP and EuroPLoP conferences, and thus went through its quality assurance process. In fact, the idea for this book was born at a security pattern workshop at EuroPLoP 2002. This guarantees that the book covers the collective experience of world-leading security experts, rather than the ideas and experiences of a sole and possibly novice individual.

To discuss patterns and pattern-related issues, the pattern community also offers several mailing lists and a World Wide Web page. The URL of the pattern home page is:

```
http://www.hillside.net/patterns/
```

This page provides useful information about forthcoming pattern events and available books on patterns, and offers references to other Web pages about patterns. There are also several Internet mailing lists on patterns. You can find their details and information on how to subscribe to them on the patterns home page shown above.

The unofficial steering committee of the pattern community is Hillside Incorporated, also known as the 'Hillside Group,' and its European arm Hillside Europe e.V. The Hillside Group is a non-profit organization made up of leading software experts: its main goal is to propagate the use of patterns in software development, to lead the pattern community, and to give support to newcomers in this new discipline of software engineering. The 'spiritual father' of the Hillside Group is Kent Beck. The Hillside Group also organizes and sponsors the PLoP conference series.

By joining the pattern community you can take advantage of all this experience, captured in many well-documented patterns that are ready for practical use. You will also be able to share your own experience in software development with other experts by writing your own patterns. We thus invite you to join the pattern community if you are not already part of it. Visit the pattern home page, subscribe to the pattern mailing lists, look at the various pattern books, attend the PLoP conferences, capture your own experience as patterns and share them with experts from all over the world. You will certainly be rewarded by many positive 'Aha!' effects, just as we were when we first discovered patterns.