1 On Patterns and Pattern Languages

Neither can embellishment of language be found
without arrangement and expression of thoughts,
nor can thoughts be made to shine
without the light of language.

Marcus Tullius Cicero, Roman stateman, orator,
and philosopher, 106–43 BC

In this chapter we introduce patterns briefly, including their history, along with a number of pattern concepts. We examine the anatomy of a pattern, what it offers, and what drives it. We explore the relationships we often find between patterns. We conclude with a discussion of pattern languages, what they are, and how they can be presented and used.

1.1 Patterns Introduced

From a design perspective, software is often thought of in terms of its parts: functions, source files, modules, objects, methods, classes, packages, libraries, components, services, subsystems, and so on. These all represent valid views of the different kinds and scales of units of composition with which developers work directly. These views focus on the parts, however, and de-emphasize the broader relationships and the reasoning that make a design what it is. In contrast, patterns have become a popular and complementary way of describing and evolving software designs, capturing and naming proven and common techniques. They emphasize the *why*, *where*, and *how* of designs, not just the *what*.

A pattern documents a recurring problem–solution pairing within a given context. A pattern, however, is more than either just the problem or just the solution structure: it includes both the problem and the solution, along with the rationale that binds them together. A problem is considered with respect to conflicting forces, detailing why the problem *is* a problem. A proposed solution is described in terms of its structure, and includes a clear presentation of the consequences—both benefits and liabilities—of applying the solution.

The recurrence of patterns is important—hence the term *pattern*—as is the empirical support for their designation as patterns. Capturing the commonality that exists in designs found in different applications allows developers to take advantage of knowledge they already possess, applying familiar techniques in unfamiliar applications. Of course, by any other name this is simply 'experience.' What takes patterns beyond personal experience is that patterns are named and documented, intended for distilling, communicating, and sharing architectural knowledge.

From Building Architecture to Software Architecture

Although patterns are now popular and relatively widespread in the world of software development, they originated in the physical world of building rather than the virtual world of software. Throughout the 1960s and 1970s the architect Christopher Alexander and his colleagues identified the concept of patterns for capturing

architectural decisions and arrangements [Ale79] [AIS77]. In particular, they wanted to focus on patterns that were 'whole'—proven solutions drawn from experience that helped to improve the quality of life for people whose environment could be shaped by such patterns. Identifying architectural patterns was a response to the perceived dysfunctionality of many popular but unsuccessful trends and practices in contemporary building architecture.

The patterns they documented were combined as a coherent set, a *pattern language* that embraced different levels of scale—the city, the neighborhood, the home—with a connected view of how to apply one pattern in the presence of another. This approach first made its way into software in the form of a handful of user-interface design patterns by Kent Beck and Ward Cunningham [BeCu87], which included suggestions of a larger pattern language for object-oriented programming. Related ideas emerged over the following years, from Jim Coplien's book on C++ idioms and styles [Cope92] to Erich Gamma's thesis on framework design [Gam92], and Bruce Andersen's vision of a *Handbook of Software Architecture*. Following this groundswell of interest and convergence of thinking, the Gang-of-Four's[1] seminal *Design Patterns* book [GoF95] was published.

In addition to the widespread interest in software patterns, a whole community has sprung up focusing on collecting and documenting patterns, improving them through shepherding and writer's workshops. Many patterns have been published online, in journals, and in books following the early work. One series of books has been the *Pattern Language of Program Design* series [PLoPD1] [PLoPD2] [PLoPD3] [PLoPD4] [PLoPD5], which are distillations from the PLoP conferences held around the world. Another is the *Pattern-Oriented Software Architecture* volumes. The first volume, *A System of Patterns* [POSA1], was complementary to *Design Patterns* in that some of its patterns built on or extended the Gang-of-Four patterns. It was also complementary in the sense that it focused more explicitly on different levels of scale in software architecture, with an emphasis on large-scale systems. The second volume in the series, *Patterns for Concurrent and Networked Objects* [POSA2], and the third, *Patterns for Resource Management*

1 The 'Gang of Four,' as they and their book have become known, are Erich Gamma, Richard Helm, Ralph Johnson, and the late John Vlissides.

[POSA3], explored specific topics of architecture—particularly concurrency and networked applications—in greater detail. The fourth volume, *A Pattern Language for Distributed Computing*, is in your hands.

1.2 Inside Patterns

Consider the following situation. You check the cupboard: it's bare. You check the refrigerator: the only thing going on is the light. It is therefore time to go shopping—you need milk, juice, coffee, pizza, fruit, and many of the other major food types. You go to the supermarket, enter, find the milk, pick some up, pay for the it, return home, and put the milk in the fridge. You then go back to the supermarket, enter, find the juice, pick some up, pay for it, return home, and put the juice in the fridge. Repeat as necessary, until you have everything you need.

Although shopping in this way is possible, it is not a particularly effective approach—even if the supermarket is next door to you. There are wiser ways to spend your time. Significant performance improvements will not come about from tweaks and micro-optimizations, such as leaving the front door unlocked, leaving the fridge door open, using cash rather than card. Instead, a fundamentally different approach to shopping is needed.

Take 2: you make out a shopping list, go to the supermarket, enter, collect a shopping cart, find each of the items on the shopping list and place them in your cart (less the ones that are out of stock, plus some impulse buys), pay for the items in the cart, return home, unpack, and put the items away.

A Problem in a Context

The domain of groceries may at first seem far removed from the domain of software development, but what we have just described is essentially a distributed programming problem and its resolution. The programming task is that of *iteration*: traversing a collection to fulfill a particular objective. When the code performing the iteration is

collocated in the same process as the collection object being accessed, the cost of access is minimal to the point of irrelevant. A change of situation can invalidate this assumption, however, along with any solution that relies on it. A distributed system introduces a significant overhead for any access—any design that does not take this context into account is likely to be inefficient, inflexible, and inadequate. This observation is generally true of patterns and design: context helps to frame and motivate a particular problem, and any solution must be sensitive to this.

Forces: The Heart of Every Pattern

Where the context sets the scene for a problem, the forces are what characterize it in detail. Forces determine what an effective solution must take into account. For any significant design decision, forces inevitably find themselves in conflict.

Returning to the context of distributed computing and the problem of traversing a collection, we are presented with a number of issues that must be considered. There is a significant time overhead in communicating over a network (or going out of the house to visit the supermarket), so a time-efficient solution cannot assume that the overhead is negligible. A further space and time overhead is involved in communication, because values (or groceries) must be marshaled and unmarshaled (packed and unpacked). We must also, however, consider convenience: picking up a carton of milk, paying for it, and putting it in the fridge involves much less ceremony than taking a shopping cart around the supermarket and packing and unpacking bags. Fine-grained iteration is more familiar to programmers, better supported by libraries and languages, and leads to less apparent blocking. We must also consider partial failure (or partial success): networks can become ‘notworks.’

Solutions and Consequences

An effective solution needs to balance the forces that make the problem a problem. It should also be described clearly. In this particular case, our solution involves preparing and sending a bulk request (the shopping list) from one address space to another, implemented with respect to the appropriate protocol (driving, walking,

cycling, and so on), performing all the traversal locally in a single batch (walking around the supermarket with the shopping cart), marshaling the results back to the caller and unmarshaling them into the local address space (paying, packing, and returning home), and then traversing the batch result locally (unpacking and putting everything away).

Another feature of any design decision is that it may not be perfect, in the sense that every design decision has consequences, some of which are benefits and some liabilities. An effective application of a pattern is therefore one in which the benefits clearly outweigh the liabilities, or one in which the liabilities do not even appear to come into play.

In this case, perhaps the overriding consideration is the communication-to-computation ratio, which has led to a design that minimizes the amount of communication involved (once out, once back) for the amount of work done (accessing multiple items) and ensures that partial failure does not result in partial state—all results are returned, or none at all. There is no 'free lunch,' however. Instead of a single iteration, there are many: a local iteration to prepare the call (compile the shopping list), a remote iteration to perform the computation and collect the results for each item in the request (walking round the supermarket), a local iteration to process each result (unpacking and putting away the groceries). The design style is rather idiomatic for distributed systems, but not necessarily for the more common programming experience of iteration over collections collocated in the same address space.

The Naming of Names

A pattern contributes to the software design vocabulary, which means that it must have a name. The name is the shorthand we use in conversation and the way we index and refer to the pattern elsewhere. Without a name, we find ourselves having to redescribe the essential elements of the pattern in order to communicate it to others.

The most effective names for patterns are those that identify some key aspect of the solution, which means that names are often noun

phrases. In the example examined here the name of the pattern is BATCH METHOD (302). This name helps to differentiate the pattern from other iteration patterns that are appropriate for different contexts or slightly different problems, such as ITERATOR (298)—the pattern applied in the first shopping attempt—and ENUMERATION METHOD (300)—a pattern that inverts the sense of iteration, encapsulating the loop within the collection and calling out to a piece of code for each element.

Brief Notes on the Synthesis of Pattern Form

Patterns are often recognized and used informally, acquiring a name based on a particular implementation, with the implication that any use of the key characteristics of a particular design follows that pattern. For software architects and developers to apply patterns more generally and broadly, however, a more concrete description is often needed. We have detailed the anatomy of a pattern in terms of concepts such as *context* and *forces*, but how are we to present the pattern in writing?

It turns out that there is more than one answer to this question. There are many pattern forms in common use, each with different emphases, and each with a different audience or reading style in mind. For example, full documentation of a single pattern that is focused on a pattern common to a particular programming language is often high on technical detail, including sample code. It may best be motivated through one or more examples. For long patterns, dividing the pattern into clearly titled subsections may make it more accessible to readers and potential users. Conversely, a pattern presented alongside many other patterns that are intended as general development-process guidelines is not as well served by length, detail, and code. A more summarized and less elaborate form may thus be appropriate in this case.

Whichever form is adopted for a pattern or catalog of patterns, the form should state the essential problem and solution clearly, emphasize forces and consequences, and include as much structure, diagramming, and technical detail as is considered appropriate for the target audience.

1.3 Between Patterns

Patterns can be used in isolation with some degree of success. They represent foci for discussion, point solutions, or localized design ideas. Patterns are generally gregarious, however, in that they are rather fond of the company of other patterns. Any given application or library will thus make use of many patterns.

Patterns can be collected into catalogs, which may be organized according to different criteria—patterns for object-oriented frameworks, patterns for enterprise computing, patterns for security, patterns for working with a particular programming language, and so on. In these cases what is perhaps most interesting is how the patterns relate. An alphabetical listing is good for finding patterns by name, but it does not describe their relationships.

Software architecture involves an interlocking network of many different decisions, each of which springs from, contradicts, suggests, or otherwise relates to other decisions. To make practical sense as an architectural concept, therefore, where they compete and cooperate, patterns inevitably enlist other patterns for their expression and variation.

Pattern Complements

It is all too easy to get stuck in a design rut, always applying a particular pattern for a general class of problem. Although this strategy can often be successful, there are situations in which not only is a habitual pattern-of-choice not the most effective approach, it can actually be the least effective. To paraphrase Émile-Auguste Chartier, 'nothing is more dangerous than a design idea when you have but one design idea.'

In a design vocabulary, as with any vocabulary, part of effective expression is based on breadth of vocabulary, particularly synonyms, each of which has slightly different qualities and implications. Two or more patterns may appear to solve the same or similar problems—ITERATOR and BATCH METHOD, for example. Deciding between them involves a proper appreciation of the context, goal of the

problem, forces, and solution trade-offs. In this sense the patterns are perceived as complementary because together they present the choices involved in a design decision.

Patterns can also cooperate, so that one pattern can provide the missing ingredient needed by another. The goal of this cooperation is to make the resulting design better balanced and more complete. In this sense patterns are complementary because they coexist and reinforce one another in the same design. Moreover, many patterns that might be characterized as alternatives that are in competition with each other, such as ITERATOR and BATCH METHOD, can also complement one another through cooperation.

Consider again the distributed iteration problem. By accessing a single element each time around the loop, ITERATOR on its own offers an overly fine-grained approach that is inefficient for remote collection access. BATCH METHOD, in contrast, replaces loop repetition across the network with repetition of data—passing and/or receiving collections. This works well in many cases, but for large collections, or for clients that need responsive replies, the time spent marshaling, sending, receiving, and unmarshaling can lead to unacceptably long periods of time when the client is just blocked. An alternative approach is to combine both patterns: use the basic concept of an ITERATOR as a traversal position, but instead of stepping a single element at a time, use a BATCH METHOD to take larger strides.

Pattern Compounds

Pattern compounds capture recurring subcommunities of patterns. They are common and identifiable enough to allow them to be treated as a single decision in response to a recurring problem. In distributed computing, the technique of combining ITERATOR and BATCH METHOD, for example, is one such example, to which the names BATCH ITERATOR and CHUNKY ITERATOR are often applied.

Pattern compounds are also known as *compound patterns*, and were originally known as *composite patterns*. In conversation, however, there is obvious scope for confusion between 'a composite pattern' and 'the COMPOSITE (319) pattern,' which is one of the widely known Gang-of-Four patterns.

In truth, most patterns are compound at one level or other, or from one viewpoint or other, so the concept is essentially relative to the design granularity of interest.

Pattern Stories

The development of a system can be considered a single narrative example, in which design questions are asked and answered, structures assembled for specific reasons, and so on. We can view the emergence and refinement of many designs as the progressive application of particular patterns. The design emerges from a narrative—a pattern story—that builds one pattern on another, responding to the design issues introduced or left outstanding by the previous pattern.

As with many stories, they capture the spirit, although not necessarily the truth, of the detail of what happens. Sequential ordering matters more in presenting a design and its evolution than it does in the actual evolution of a design. It is rare that our design thinking fits into a tidy, linear arrangement, so there is a certain amount of retrospection, revisionism, and rearrangement involved in retelling how a design played out over time.

These stories, then, may be of systems already built, forecasts of systems to be built, or simply hypothetical illustrations of how systems could be built. They may be recovered whole or idealized from the development of real systems, a guide to the architecture and its design rationale. They may be used as storyboarding technique for envisioning and exploring future design decisions and paths for a system. They may be speculative and idealized, intended to teach or explore design thinking, but not an actual system.

Pattern Sequences

Pattern sequences are related to pattern stories in the same way that individual patterns are related to examples that illustrate or motivate those patterns: they generalize the progression of patterns and the way a design can be established, without necessarily being a specific design. In this sense, a given pattern sequence can be considered a highly specific development process. Predecessor patterns form part of the context of each successive pattern.

For example, BATCH ITERATOR as the application of ITERATOR and BATCH METHOD can also be seen as a (very) short sequence, in which first ITERATOR is applied to provide the notion of a traversal position, then BATCH METHOD is applied to define the style of access.

1.4 Into Pattern Languages

While patterns represent a design vocabulary, pattern languages are somewhat like grammar and style. Through the use of patterns, a pattern language offers guidance on how to create a particular kind of system, or how to implement a certain kind of class, or how to fulfill a particular kind of cross-cutting requirement, or how to approach the design of a particular family of products. Whether we are interested in building a distributed system for managing a warehouse, writing exception-safe code in C++, or developing a Java-based Web application, if there is experience in the domain of interest, it is likely that this experience can be distilled into patterns and organized as a pattern language.

From Sequences to Languages

While pattern stories are concrete and linear, pattern sequences are more abstract but still essentially linear. Feedback should inform the designer how to apply the next pattern in a sequence, or whether to revisit an earlier application. Pattern languages are more abstract still, and typically more richly interconnected.

A pattern language defines a network of patterns that build on one another, typically a tree or directed graph, so that one pattern can optionally or necessarily draw on another, elaborating a design in a particular way, responding to specific forces, taking different paths as appropriate. The relationships explored in the previous section, *Between Patterns*, are those that can be found in various forms within a pattern language. For example, a pattern sequence defines a path through a language, taking in some or all of its patterns, and a pattern story recalls a route along one path.

Presenting and Using Pattern Languages

A pattern language includes its sequences, and the knowledge of how to handle feedback should be considered part of the scope and responsibility of a language. A given pattern sequence can be used as a guide to the reader about one way that a language has been, can be, or is to be used. When taken together, a number of sequences can be seen to provide guidance on the use of a given pattern language—or, alternatively, when taken together, a number of sequences can be used as the basis of a pattern language.

Pattern sequences therefore have the potential to play a number of roles. Other than in the form of pattern stories, however, they are normally not made explicit as part of the presentation of a language. As a result there is generally more discussion in the patterns community about pattern sequences than actual cataloging or specific description of them. Given that different pattern sequences give rise to different common design fragments with different properties that are useful in different situations, it seems worthwhile to document some of these, even if briefly. Of course, enumerating all the reasonable sequences for anything but a small or simply structured language is likely to be a Sisyphean task that will overwhelm both its author and any readers who try to use it for guidance.

The common vehicle for illustrating pattern languages in action is stories. Of course, there is the risk that stories may be taken too literally. In the way that a motivating example in a pattern is sometimes mistaken for the pattern itself, a pattern story may end up stealing the limelight from the language it represents. Although readers are free to generalize, they may be drawn to the specifics of an example to the exclusion of its general themes and structure.

It is therefore crucial to strike the right balance between the specific and the general to ensure that the patterns within a language are also documented in a sufficiently complete form individually, but with obvious emphasis on their interconnections. A single pattern can often be used in a variety of situations and a variety of different pattern languages. To keep its role within a language focused, it makes sense to concentrate on documenting the aspects that are relevant to the language, and reducing or omitting aspects that are only relevant in other situations. The context for a given pattern can also

be narrowed to predecessor patterns in the language. This overall mix of specific examples, in-context patterns, and relationships between patterns offers a practical approach to presentation and usage of pattern languages.

1.5 Patterns Connected

The value that individual patterns have should not be underrated, but the tremendous value that they have when brought together as a community should not be underestimated. Patterns are outgoing, fond of company, and community spirited.

It is this networking on the part of patterns that reflects the nature both of design and of designs. The notions of synthesis, overlap, reinforcement, and balance across different design elements according to the roles that they play reinforces what to some appears an initially counterintuitive view of design: that the code-based units of composition found in a given design are not themselves necessarily the best representation of the design's history, rationale, or future.