1 A Solution to a Problem and More

Philosophy, if it cannot answer so many questions as we could wish, has at least the power of asking questions which increase the interest of the world, and show the strangeness and wonder lying just below the surface even in the commonest things of daily life.

Bertrand Russell

In this chapter we take a simple problem–solution pairing and use gradual and reasoned steps to expand and refine it into a description that more obviously qualifies as a pattern description. En route we examine the different facets that a pattern embodies and that its description should contain. Our incremental approach also illustrates the iterative nature of pattern writing. The use of feedback, reflection, growth, and refinement is a normal part of the pattern-writing process, not simply an artifact of how we have chosen to illustrate the facets that comprise a sound pattern description.

1.1 A Solution to a Problem

If asked for a one-sentence characterization of what a pattern is, many pattern-aware software developers might respond with something like:

> 'A pattern is a solution to a problem that arises within a specific context.'

This seems a fair summary at first glance. It is certainly not a false one, although it may not cover the whole truth. The truth of the summary is supported by looking around at the common patterns in use today: they provide working, concrete, and adaptable solutions to problems that repeatedly arise in certain situations during software development, from organizational to programming contexts. The summary is also a good fit with most popular pattern definitions.

An Example

A pattern is more, however, than just this soundbite! If it were sufficient, the following would have all that was needed when presenting a pattern:

> **Context**: A client needs to perform actions on an aggregate data structure.
>
> **Problem**: The client may need to retrieve or update multiple elements in a collection. If the access to the aggregate is expensive, however, accessing it separately for each element over a loop can incur severe performance penalties, such as round-trip time, blocking delay, or context-switching overhead. How can bulk accesses on an aggregate object be performed efficiently and without interruption?
>
> **Solution**: Define a single method that performs the access on the aggregate repeatedly. Instead of accessing aggregate elements individually and directly from within a loop, the client invokes the method to access multiple elements in a single call.

This description is obviously a solution to a problem that arises within a specific context. But is it, as presented, a pattern in the profound sense in which the term is often applied?

Recurrence and Goodness

It is not simply enough for a solution to a problem to exist to consider a reasoned design for it a pattern. It must recur—which is, after all, the motivation behind the use of the word 'pattern,' as any dictionary will attest. In almost all cases in which the term 'pattern' is used in the context of software design, there is an underlying assumption that we are talking about 'good' patterns—that is, patterns that are complete, balanced, and clearly described, and that help to establish some kind of wholeness and quality in a design. This assumption is so common that the word 'good' is almost always dropped: the word 'pattern' effectively subsumes it.

If a pattern, however, represents a commonly recurring design story—one that varies a little with each retelling—there are not just good stories to tell. Some recurring practices are incomplete, unbalanced, and fail to establish a recognizable wholeness and quality. What they lack in goodness they often make up for in popularity—in spite of usage to the contrary, 'common practice' and 'best practice' are not synonyms [Gab96].

Some patterns have recurrence but not quality. These are 'bad patterns,' a phrasing used by Christopher Alexander in his original framing of patterns [Ale79] and the motivation for identifying 'good patterns.' 'Bad patterns' are dysfunctional, often twisting designs out of shape, leading to a cycle of 'solving the solution,' in which the ingenuity of developers is consumed in fixing the fixes (to the fixes to the fixes…).

In identifying and separating good patterns from dysfunctional patterns or nonpatterns, we first need to distinguish between solution ideas that recur and those that are singular, and so tightly bound to their application that whatever their merits as specific designs, they cannot be considered general patterns. We could say that each recurring solution theme qualifies as a 'candidate pattern.'

We also need more than just the idea of recurrence in a pattern to make it a useful part of our development vocabulary. We need to know that its application is beneficial rather than neutral or malignant—a pattern with no effect or ill effect. Before hastily branding 'dysfunctional' any candidate pattern that fails to contribute successfully to a

design, however, we must also be able to distinguish between patterns that are truly dysfunctional and patterns that have simply been misapplied—in other words, dysfunctional applications of otherwise sound patterns. To assess these qualities reasonably we need to render our candidate pattern in some form that makes them visible. A pattern description must be clear and must make it possible to see the qualities of the pattern—a poor pattern description can obscure a good pattern, selling it short, whereas a good 'spin' can mask a poor pattern, overselling it [Cool98].

Looking back at our original example above, we can see that it qualifies as a 'candidate pattern,' but that those additional qualities of interest are still not quite visible in its current form. Until we can be confident of the pattern's viability and the ability of our documentation to convey that, we should consider it a 'proto-pattern.' So what is missing from our proto-pattern description that might confirm its candidacy as a good pattern?

1.2 A Process and a Thing

When analyzing the content of our example proto-pattern, perhaps the most glaring deficiency is the vagueness of its solution part: it describes a process for creating a solution, but does not suggest what concrete solution to create. Although a pattern can be realized a 'million different ways,' [Ale79] this diversity arises from the precise detail of an actual problem and its context, not from the vagueness of a proposed solution structure. A pattern should inform its reader how to build a solution in a way that is loose enough to be general, but tight enough to avoid vagueness and accidental ambiguity.

Many solution implementations are possible for our example proto-pattern: all could be said to follow its solution process, but few could be said to follow the spirit of the pattern's message. For example, one valid implementation would be to define a method in the client that contained the loop. This design would be a simple refactoring within the client code, extracting one method from another, but having no effect whatsoever on the issues highlighted in the problem statement.

Another valid implementation would be to define a single method on the aggregate object and have it return each individual result to the client by callback, in the style of a push notification. Such an implementation, however, would not be very practical, presenting more of a problem than a solution in its costly accumulation of round-trip time. It might follow the letter of the proposed solution, but it would not follow its spirit. For the original problem there are many known (and better) solution paths: it is the job of the pattern description to document its chosen route with accuracy and precision.

The solution part of a good pattern describes both a *process and a thing*: the 'thing' is created by the 'process' [Ale79]. For most software patterns, 'thing' means a particular high-level design outline or description of code detail. There is an important distinction between this understanding of the term 'process' and the idea of 'process' as a full-lifecycle prescription of software development. The popular and traditional notion of software development process targets the development of a whole system, from its inception to its deployment. It is ambitious in scale, general in nature, alone in its application.

An individual pattern, by contrast, is focused on solving a single specific problem. It is unambitious but determined, with well-defined boundaries and a generality that arises from depth rather than breadth. On a project, an individual pattern will enjoy the company of other patterns rather than sit on its own. A pattern might be considered a microprocess—or even nanoprocess—against the backdrop of a full-lifecycle software development process.

An Example (Take 2)

Our proto-pattern's original solution part is missing the concrete 'thing' to build. We can revise that paragraph to arrive at a second version:

> Define a single method that performs access on the aggregate repeatedly. This method is defined as part of the interface of the aggregate object. It is declared to take all the arguments for each execution of the action, for example via an array or a collection, and to return results by similar means. Instead of accessing aggregate elements individually and directly from within a loop, the client invokes the method to access multiple elements in a single call.

This second solution version is much clearer than the original one: we now know *what* concrete solution to build to resolve the given problem, not just *how* to build this solution.

1.3 Best of Breed

Now that our revised solution statement gives us a better idea of what to build, we can more clearly see the mismatch between what is described as a problem and what is proposed as a solution. Although the recommendation is to encapsulate all repeated accesses of an aggregate object in a single method, the problem statement says the following:

> The client may need to retrieve or update multiple elements in a collection. If the access to the aggregate is expensive, however, accessing it separately for each element over a loop can incur severe performance penalties, such as round-trip time, blocking delay, or context-switching overhead. How can bulk accesses on an aggregate object be performed efficiently and without interruption?

This statement does *not* say that there is only one form of access or action on the aggregate object. Rather it refers to bulk accesses in general, of which retrieval and update are obvious and representative examples. If we implement the second solution version, however, we would find that all bulk accesses would be channeled through a single method. We would have to include some additional parameter or parameters to accommodate the variation in the actions that might be performed against the aggregate object. This method would be weakly cohesive, supporting a superset interface for any data that might flow to and from the aggregate, and some kind of selector to indicate which action was required. Such a method would be tedious and error-prone to use and maintain.

SINGLETON is an example of a well-known pattern with a weak solution. Its original intent is to 'ensure a class only has one instance, and provide a global point of access to it' [GoF95]. The solution proposed to resolve this problem is to disallow creation of an instance of a class via its constructor. Instead, the pattern introduces a *globally* accessible

method that creates and returns a new instance of the class on demand: if a class-level static reference to a singleton instance is null, a new instance is created and returned, otherwise the reference to the existing singleton instance is returned.

The problem with this solution is the use of a single global access point to an object, which is considered poor practice in modern programming, whether in enterprise or embedded systems, since it tightly couples code that uses the singleton class to a particular context, inappropriately hardwiring the architecture to a set of potentially unstable assumptions. Consequently, a number of issues arise when dealing with SINGLETON, including (but not limited to):

- How to make a SINGLETON thread-safe?
- How to customize SINGLETON behavior?
- How to dispose of or exchange a SINGLETON instance?
- How to unit-test code dependent on a SINGLETON?

The literature, such as [Vlis98b] [BuHe03] [POSA2], that discusses these issues dwarfs the page count of the original pattern description in the Gang-of-Four book! Working around or fine tuning SINGLETON appears to be as popular a sport as introducing it into a system. Deconstructing the SINGLETON pattern, therefore, suggests that it may introduce more problems than it resolves—a fact that Kent Beck summarized nicely [Beck03]: 'How do you provide global variables in languages without global variables? Don't. Your programs will thank you for taking the time to think about design instead.'

In practice a useful pattern should therefore not propose just any solution for the problem it is addressing. Instead, it should present a robust solution that resolves the problem optimally. The solution in a *good* pattern, moreover, needs to have a proven track record. Quality patterns do not represent neat ideas that *might* work, but concepts that have been applied repeatedly and successfully in the past—this recurrence is what makes a pattern a pattern, and the track record is what demonstrates its quality.

The Gang-of-Four puts it this way: 'Patterns distill and provide a means to reuse the design knowledge gained by experienced practitioners' [GoF95]. Brian Foote put it even more succinctly: 'Patterns are

an aggressive disregard of originality' [PLoPD3]. Consequently, new ideas must first prove their worth in the line of active duty—often many times—before they can truly be called patterns.

An Example (Take 3)

Another revision of our proto-pattern solution provides us with a third version—which has the quality we expect from a 'good' pattern:

> For a given action, define a single method that performs the action on the aggregate repeatedly. This method is defined as part of the interface of the aggregate object. It is declared to take all the arguments for each execution of the action, for example via an array or a collection, and to return results by similar means. Instead of accessing aggregate elements individually and directly from within a loop, the client invokes the method to access multiple elements in a single call.
>
> Each method folds repetition into a data structure rather than a loop within the client, so that looping is performed before or after the method call, in preparation or follow-up. Consequently, the cost of access to the aggregate is reduced to a single access, or a few 'chunked' accesses. In distributed systems this 'compression' can significantly improve performance, incur fewer network errors, and save precious bandwidth.
>
> The trade-off in complexity is that each method performs significantly more housekeeping to set up and work with the results of the call. This solution also requires more intermediate data structures to pass arguments and receive results. The higher the costs for networking, concurrency, and other per-call housekeeping, however, the more affordable this overhead becomes.

This third version better resolves the original problem, avoiding the problems identified for the second version.

1.4 Forces: the Heart of Every Pattern

The previous discussion reveals that the problem addressed by our proto-pattern is not as easy to resolve as it might first appear. The pure problem cannot be considered in isolation. There are also a number of requirements on its solution—for example, that it should

be bandwidth-friendly. It is hard—if not impossible—to derive such requirements from the simple problem statement, however. Yet to achieve the desired quality of implementation, the problem's solution needs to address them.

If, on the other hand, such requirements were added to the proto-pattern's problem statement, they would become explicit, and a concrete solution for the problem could deal with them appropriately. The same argument holds for any desired property that the solution should provide. Similarly, there may be some constraints that limit the solution space for the problem, or just some things worth taking into account when resolving it. These requirements, desired properties, constraints, and other facts fundamentally shape every given problem's concrete solution. The pattern community calls these influencing factors *forces*, a term taken from Christopher Alexander's early pattern work [Ale79].

Forces tell us why the problem that a pattern is addressing is actually a problem, why it is subtle or hard, and why it requires an 'intelligent'—perhaps even counter-intuitive—solution. Forces are also the key to understanding why the problem's solution is as it is, as opposed to something else. Finally, forces help prevent misapplications of a pattern in situations for which it is not practical [Bus03a].

For example, much of the confusion about SINGLETON could have been avoided if the original description [GoF95] had listed a force like 'The property of having only one instance of class is a property of the *type* being represented, such as a class that realizes a stateful API to a specific piece of hardware in a system, and not simply an incidental property of an *application* that uses only a single instance of a class.' SINGLETON would still have a place in the pattern space, but not the broad and overbearing dominion it has now—which is a root cause for the trouble this pattern can cause [Ada79].

An Example (Take 4)

To motivate the concrete solution proposed by our proto-pattern, we therefore add the following four forces [Hearsay01] to its description:

> An aggregate object shared between clients in a concurrent environment, whether locally multithreaded or distributed, is capable of encapsulating synchronization for individual method calls but not for multiple calls, such as a call repeated by a loop.
>
> The overhead of blocking, synchronization, and thread management must be added to the costs for each access across threads or processes. Similarly, any other per-call housekeeping code, such as authorization, can further reduce performance.
>
> Where an aggregate object is remote from its client, each access incurs further latency and jitter, and decreases available network bandwidth.
>
> Distributed systems are subject to partial failure, in which a client may still be live after a server has died. Remote access during iteration introduces a potential point of failure for each loop execution, which exposes the client to the problem of dealing with the consequences of partial traversal, such as an incomplete snapshot of state or an incompletely applied set of updates.

A quick check reveals that the third version of the solution statement already satisfies the first three forces, either directly or as a by-product of the proposed design. In particular, synchronization is isolated to a single method call rather than across many, the concurrency cost is reduced by replacing multiple calls with a single call, and the remote access cost is reduced by replacing multiple calls with a single call. The fourth force is not yet addressed by this third solution version, however, since the result of a failed call could yield a partial update or an incomplete query. We noticed this deficiency only because we thought hard about the forces and made them explicit. We also noticed that we can afford to make some other consequences more explicit.

These considerations lead to a fourth version of the solution:

> For a given action, define a single method that performs the action on the aggregate repeatedly.

> This method is defined as part of the interface of the aggregate object. It is declared to take all the arguments for each execution of the action, for example via an array or a collection, and to return results by similar means. Instead of accessing aggregate elements individually and directly from within a loop, the client invokes the method to access multiple elements in a single call.
>
> Each method folds repetition into a data structure rather than a loop within the client, so that looping is performed before or after the method call, in preparation or follow-up. Consequently, the cost of access to the aggregate is reduced to a single access, or a few 'chunked' accesses. In distributed systems this 'compression' can significantly improve performance, incur fewer network errors, and save precious bandwidth.
>
> Each access to the aggregate becomes more expensive but the overall cost for bulk accesses has been reduced. Such accesses can also be synchronized as appropriate within the method call. Each call can be made effectively transactional, either succeeding or failing completely, but never partially.
>
> The trade-off in complexity is that each method performs significantly more housekeeping to set up and work with the results of the call. This solution also requires more intermediate data structures to pass arguments and receive results. The higher the costs for networking, concurrency, and other per-call housekeeping, however, the more affordable this overhead becomes.

Now all four forces are resolved and the resulting solution is stronger. Even though the second and third force were addressed together, it is still important to list them as separate forces: they are different and they are independent. Network bandwidth is the not the same kind of resource or consideration as context switching cost: the communication cost may vary independently from platform processing power. This variability underscores the importance of listing all forces explicitly, even if at first glance they appear superfluous.

Unfortunately it might not always be possible to resolve all forces completely in a solution: forces are likely to contradict and conflict with one another. A particular force may address an aspect that can only be resolved at the expense of the solution's quality in respect to aspects addressed by other forces. For example, efficiency can often

be achieved only by giving up flexibility, and vice versa. In such cases a solution must *balance* the forces, so that each is resolved sufficiently, if not completely, to meet the requirements sufficiently.

Dysfunctional, Bad, or Anti?

Having discussed forces, we now are in a position to revisit and clarify some distinctions in terminology. Christopher Alexander favored characterizing recurring problematic approaches as *bad* patterns [Ale79], whereas our emphasis has been to characterize them as *dysfunctional* patterns. You may also have come across the term *anti-patterns*, which also sounds applicable.

The term 'anti-pattern' has quite a sensational ring to it, since it sets up some kind of contrast or conflict. It is not immediately clear, however, what the term entails. In looking at other words that follow the 'anti-' pattern, there are a number of possibilities. For example, in the manner of antibiotics and antidotes, perhaps they are a cure for someone with 'patternitis'—a term often used to describe someone who has become too carried away in their use of patterns.

Perhaps anti-patterns provide a cosmic balance for patterns, canceling them out in some dramatic way, as matter and antimatter do on contact? Perhaps they are simply against patterns, in the manner of many political 'anti-' slogans, or they are a way of bringing them down, in the manner of anti-aircraft devices? Perhaps they precede a deeper understanding of patterns, as antipasto precedes a main course? Perhaps they offer only a disappointing conclusion, an anticlimax? Jim Coplien [Cope95] offers the following clarification:

> *Anti-patterns* are literature written in pattern form to encode practices that don't work or that are destructive. Anti-patterns were independently pioneered by Sam Adams, Andrew Koenig [Koe95], and the writer. Many anti-patterns document the rationalizations used by inexpert decision makers in the Forces section. […]
>
> Anti-patterns don't provide a resolution of forces as patterns do, and they are dangerous as teaching tools: good pedagogy builds on positive examples that students can remember, rather than negative examples. Anti-patterns might be good diagnostic tools to understand system problems.

This description was written before the first *Antipatterns* book [BMMM98] popularized the term. In that book the notion of anti-pattern is at times ambiguous: sometimes *anti-pattern* refers to a recurring problem situation—a failing solution—that results from misapplied design practices. At other times *anti-pattern* also appears to include a solution response to these problem situations.

The former definition appears closer to the medical metaphor of diagnoses [Par94] [EGKM+01] [Mar02a] [Mar02b] [Mar03], identifying a problem situation through its symptoms and root causes. The latter approach also appears to fit this metaphor, with a proposed remedy and prognosis, but it also fits the basic pattern concept: the failing solution is the problem and the cure is the solution. In this latter form there appears nothing special—let alone 'anti'—about so-called anti-patterns as solutions to problems. The ambiguity lingers, however, which motivates the need for a clearer, less affected term for discussing recurring, problematic approaches.

The definition provided by Jim Coplien seems both the most useful and the most cautionary. Labeling the problem of unresolved forces with an 'anti' is a little ambivalent and does not properly communicate the problematic nature of such recurring solution attempts. By contrast, the term 'bad patterns' pulls no punches. As a characterization, however, partitioning designs into 'good' and 'bad' can seem a little simplistic or even sententious. This kind of value judgment may be meaningful for an individual, but does not necessarily encourage open and balanced discussion with others.

With patterns we seek to communicate and understand, so drawing on a more neutral vocabulary seems prudent. For this reason, we favor characterizing successful solutions that address the forces in a problem adequately as 'whole' and those that do not as 'dysfunctional' [Hen03a]. The former term entails balance and completeness—whole as opposed to half-done or half-baked—whereas the latter suggests that a design is impaired in some way and may undermine the stability of a larger whole.

1.5 The Context: Part of a Pattern or Not?

Now that the forces supplement the problem statement, and the solution for the problem is adjusted accordingly, we can turn our attention to the context part of the proto-pattern:

> A client needs to perform actions on an aggregate data structure.

This is a fairly general context. It probably gives rise to a number of different problems, not just to the one that the example proto-pattern addresses. In fact, it is so general that little is lost if it is dropped completely.

The context plays an important role in a pattern, however, since it defines the situation to which it applies. The more precisely this situation is described, the less likely it is that developers will use the pattern inappropriately. Ideally, the context should describe only the particular situation that can lead to the problem for which the pattern specifies a solution.

A context that is too broad, in contrast, will run the risk of making a pattern a jack-of-all-trades but a master of none. One example of such a pattern is BRIDGE, whose original intent in [GoF95] is: 'Decouple an abstraction from its implementation so that the two can vary independently.' Many designs can benefit from this level of indirection, ranging from making implementation bindings more flexible to writing C++ code with strong exception-safety guarantees [BuHe03].

Each of the many possible problems addressed by BRIDGE, however, has its own context, such as 'we are creating a component whose implementation must be exchangeable at runtime' or 'we are implementing a C++ application that must be exception-safe.' These differences result in corresponding differences in the forces associated with the problem and also differences in the concrete solutions proposed to resolve them. Remember, a pattern is often cast as a solution to a problem that arises in a specific context—which is the (incomplete) 'definition' for patterns from the beginning of this chapter. Consequently, lacking a precise context, a pattern can become all things to all people, with each having their own different—and often incompatible—view [BuHe03].

An Example (Take 5)

To sharpen the context of our proto-pattern, we merge the first two sentences of its problem statement—which already provides some context information—with the original context, and also add information about the design activity and the corresponding application's usage that lets the problem arise:

> In a distributed system, the client of an aggregate data structure may need to perform bulk actions on the aggregate. For example, the client may need to retrieve all the elements in a collection that have specific properties. If the access to the aggregate is expensive, however, because it is remote from the client, accessing it separately for each element over a loop can incur severe performance penalties.

This context is much more precise in describing the situation in which the problem arises. The context tells us where our proto-pattern may be applicable and, by implication, where it may not be applicable. Based on an understanding of the context, the developer can actively decide *not* to introduce the design. For example, if the aggregate object is not remote from the client, an alternative approach may be preferable, such as one of the solutions that we mentioned before modifying the original solution for the first time (see *page 32*). In this particular case, the proto-pattern's current solution idea for minimizing execution overhead would be like 'shooting mosquitoes with a machine gun.'

Moving information from the problem to the context statement requires rephrasing the remaining part of the problem description so that it becomes meaningful again:

> How can bulk accesses on an aggregate object be performed efficiently and without interruption if access is costly and subject to failure?

As a side effect of removing context information from the problem statement, the 'real' problem shines through more clearly. It is briefer and crisper, and we understand more directly that this is a problem.

Context Generality

Narrowing the generality of the original context has the benefit of being more precise… but it also has a drawback. There may be other situations in which the problem addressed by the example proto-pattern can arise, and where the same solution helps to resolve the problem. Consider, for example, the following context:

> In an application with a custom, in-memory database of complex—as opposed to relational—objects, separate key-based searches for individual objects are possible but potentially a bottleneck when performed repeatedly. However, certain well-defined actions, each of which returns a result, must frequently be performed on all objects in the database that satisfy a specified search criterion.

Knowing that there may be even more such situations gives rise to a challenge: how can we ensure the context's completeness? An overly general context that acts as an umbrella for many possible problem situations may be too vague, leading to inappropriate applications of a pattern. On one hand, too much detail will probably yield an unreadable pattern prefixed by an exhaustive 'laundry list' of specific context statements. On the other, an overly restrictive context may prevent developers from applying a pattern in other situations in which it is also applicable. They may take the pattern only as written and fail to grasp the opportunity for generalization it implies.

One way of coping with this problem is to start with a context that describes the known situations in which a pattern is useful, and to update this context whenever a new appropriate situation is found. The context section would then look and feel similar to the *Applicability* section of the Gang-of-Four pattern description form [GoF95], where the Gang-of-Four listed all the specific situations in which they knew the pattern could be and had been applied.

Another way to resolve this problem is to follow the approach taken in the second volume of the *Pattern-Oriented Software Architecture* series, *Patterns for Concurrent and Networked Objects* [POSA2]. There the focus of the context statements was narrowed to the theme of the book: concurrency and networking. Each pattern's context addressed situations related only to these topics, and the patterns' applicability in other situations was addressed in a separate chapter.

A narrow approach to contexts works well if a collection of patterns is centered around a common theme: the context attached to each pattern is lean, so readers can readily identify applicability in a particular domain. Other situations in which the patterns may help are not stressed, although they are not forgotten. The patterns benefit from being focused, narrowed with respect to context, rather than trying to be all things to all people.

Unfortunately, neither approach truly resolves the context-completeness problem: overlooking a situation in which a pattern is applicable is quite likely. As a result, the overly general contexts in *A System of Patterns* and this section's example proto-pattern, the split contexts in *Patterns for Concurrent and Networked Objects*, and the applicability section of the Gang-of-Four patterns [GoF95] do not work very well. On the other hand, the two latter approaches seem more practical than a general and possibly vague context. It is better to support the appropriate application of a pattern in a few specific situations than to address the possible application of a pattern everywhere—a jack of all trades, but a master of none.

Context Detached

A completely different approach is to consider the context as *not* being a part of an individual pattern at all. We have already shown that multiple contexts are possible for the example proto-pattern, but there is only one problem and one solution statement. Another possible perspective is that contexts are needed only for describing pattern languages, where they are used to specify how the languages integrate their constituent patterns. In other words, the contexts define a network of patterns—the pattern language—but the nodes—the individual patterns—are independent of the network, hence they are context free. Consequently, the context-completeness problem appears not to arise. If there are multiple pattern languages in which a pattern is useful or needed, each will provide its own context for the pattern.

While this extrinsic approach seems tidy, it gives rise to another problem. If a pattern does not characterize the situation in which a problem occurs, how can the pattern be said to honestly characterize the nature of the problem and its scope? Problems are not context-free modular parts that can simply be plugged into any context. Context

is not simply glue: there is the implication of fit, not simply adhesion, between context and problem. Thus characterization of a problem implies characterization of the context.

To some extent the question of context is a question of perspective. From the perspective of an individual pattern it is important to know the situations in which it can be applied successfully. The question of how a pattern is integrated with other patterns is useful, but less important. So the context is probably part of a pattern, which, unfortunately, allows the context-completeness problem to arise.

From a pattern language perspective it is necessary to know how the language's constituent patterns connect. It is not necessary to know which other pattern languages also include a particular pattern. The context is therefore only needed to define a specific pattern language, not to properly describe a particular pattern and the many situations to which the pattern applies. Under this characterization, therefore, the context-completeness problem appears to go into remission. This simplification, however, comes at the expense of limiting the scope of applicability of the underlying pattern. This tension and interplay between the general and the specific is an enduring and (appropriately) recurring theme in our exploration of the pattern concept.

This chapter is about stand-alone patterns, not pattern languages, so it takes the patterns-eye view: some or all of the context is part of a pattern. This view also corresponds to the majority of the software patterns, which stand alone and are not yet organized into pattern languages. We will return to the question of context in later chapters that focus on more than stand-alone patterns.

An Example (Take 6)

We can revise the context of our proto-pattern such that it captures and characterizes the known situations in which it applies and which are of interest—in this case systems with an element of multithreading, distribution, or both:

> In a distributed or concurrent system, the client of an aggregate data structure may need to perform bulk actions on the aggregate. For example, the client may need to retrieve all elements in a collection that have specific properties. If the access to the aggregate is expensive, however, because it is remote from the client or

> shared between multiple threads, accessing it separately for each element over a loop can incur severe performance penalties, such as round-trip time, blocking delay, or context-switching overhead.

Independent of the 'context-completeness question,' the proto-pattern has improved once again. Developers can get a good picture of the situations in which it may be applied, even if the new context does not enumerate all possible situations.

1.6 Genericity

With the revised context, problem, and solution sections described above, the example proto-pattern looks much more like a useful and beneficial pattern than the version with which we started. We are still not done, however. Although the solution section is quite specific with respect to its use of objects and methods, this might not be our intention. In particular, is the essence of the pattern solution being overly constrained? Is the solution offered at the same level as the problem to be solved?

Specific problems with object structure should be addressed using the tools and concepts of object orientation. A problem expressed more generally, however, should not suddenly be shackled to an object-oriented frame of reference. Conversely, patterns that deal with programming language specifics in the problem must also articulate their solutions at that level—anything more general will appear vague and imprecise. In general, patterns are as independent or dependent on a particular implementation technology as is necessary to convey their essence. For example, the problem our proto-pattern addresses can arise in procedural code, not just in object-oriented code. It is possible to express the solution for this problem in the context of objects and methods, plain functions, wire-level protocols, or even overseas parcel delivery. These alternatives do not violate the solution's core principle.[9]

9. Even patterns that seem to depend on a specific implementation paradigm, for example object technology, often do not: PROXY, for example, loses little of its essence by giving up inheritance, while STRATEGY can be implemented in C by using function pointers instead of object-level polymorphism.

If the problem addressed by a pattern requires the presence or the use of a specific technology or paradigm, this dependency should be stated explicitly. If not, a conscious decision should be taken as to whether narrowing the description in this way benefits the intended audience. Otherwise the pattern should not depend on the technology or paradigm, to avoid overly constraining its applicability and implementation options.

An Example (Take 7)

Let us assume that our goal was to provide an object-oriented solution for our problem. This aspect can be 'fixed' either by adding another force to the proto-pattern's description, if the use of object technology is a requirement for resolving the problem, or by extending its context, if using object technology is a precondition in our situation. We decide on the latter and change the first and last sentences of the context statement:

> In a distributed or concurrent object system, the client of an aggregate object may need to perform bulk actions on the aggregate. For example, the client may need to retrieve all elements in a collection that have specific properties. If the access to the aggregate is expensive, however, because it is remote from the client or shared between multiple threads, accessing it separately for each element over a loop—whether by index, by key, or by ITERATOR—can incur severe performance penalties, such as round-trip time, blocking delay, or context-switching overhead.

These changes may not be the best way to indicate that object technology should be used to resolve the problem addressed by our example proto-pattern, but at least the assumption is now explicit and the wording more precise. This discussion also leads us to another consideration: the aggregate object's interface. Consider the second sentence of the current solution section:

> Each method is defined as part of the interface of the aggregate object.

What does this imply about the aggregate object? Does it mean that whatever interface is used to declare its capabilities must include a declaration for the new method? Or are other schemes possible? For example, a separate EXPLICIT INTERFACE that offers just the iteration

capability can be defined and used as a commodity to specify the same capability in other object types, allowing the uniform treatment of many different aggregate types via a common type.

And what of the aggregate's implementation? The proto-pattern currently has nothing to say on the matter. It may be that the aggregate's class implements the new method directly. It is also feasible, however, that the underlying aggregate mechanism is left untouched and an OBJECT ADAPTER is used instead, adapting the mismatched interfaces of a distribution-friendly model and an in-process implementation.

A generic problem resolution does not necessarily introduce specific classes, components, or subsystems. Instead, it introduces *roles* [RWL96] [Rie98] [RG98] [RBGM00] that particular components of the system must adopt to resolve the original problem well. A role defines a specific responsibility within a system, and includes interaction with other roles.

Although separating the implementations of different roles is recommended, encapsulation in separate, distinct components is neither required nor implied. A single component may take on multiple roles. Roles can be assigned to existing components or they can be expressed in new components. If necessary, developers can introduce role-specific interfaces for components, so that clients see only the roles they need.

Roles are key to the seamless and optimal integration of a pattern into an existing software architecture, and for combining multiple patterns in larger-scale designs. We therefore revise the proto-pattern's solution part once again, producing a new version:

> For a given action, define a single method that performs the action on the aggregate repeatedly.
>
> Each method is defined as part of the interface of the aggregate object, either directly as part of the EXPLICIT INTERFACE exported for the whole aggregate type, or as part of a narrower, mix-in EXPLICIT INTERFACE that only defines the capability for invoking the repeated action. The method is declared to take all the arguments for each execution of the action, for example via an array or a collection, and to return results by similar means. Instead of accessing aggregate elements individually and directly from within a loop, the client invokes the method to access multiple elements in a

> single call. The method may be implemented directly by the aggregate object's underlying class, or indirectly via an OBJECT ADAPTER, leaving the aggregate object's class unaffected.
>
> [...]

Now developers can declare the method for accessing the aggregate as part of an existing interface or in an interface of its own and, similarly, implement the method as part of an existing class or in a class of its own. The roles within the proto-pattern are stable but accommodate variation in implementation.

The solution part of a pattern should therefore state explicitly which particular roles must be implemented in their own components. For all other roles it introduces, a pattern should not prescribe an implementation that unnecessarily restricts the solution's genericity.

The latest solution version still describes how to construct a structure that resolves the original problem. With roles, however, there are many more implementation choices available for the solution. Roles thus contribute significantly to the genericity of a pattern—much more than a strict class approach can ever do. Roles also make it easier to adapt a pattern's core idea to the needs of a concrete application. Unnecessary complexity and indirection levels can be avoided, which leads to simpler, more flexible, and more effective pattern implementations.

1.7 A Diagram Says More than a Thousand Words... or Less

Now that the latest solution version provides more of the qualities we expect from a whole pattern, it is worth spending some time discussing a pattern's solution part more generally. Abstracting from its concrete look-and-feel, the solution part of a software pattern commonly specifies a design structure consisting of roles that are connected through various relationships. The structure is completed by some behavior that 'happens' in this structure.

The solution part of a code-centric pattern typically offers a code fragment in which designated elements of a programming language are arranged in a specific way, together with this code's behavior.

For organizational patterns, the solution part introduces a particular organizational structure, roles in this structure, their responsibilities, and the communication between the roles. Speaking most generally, a pattern is often said to define a spatial configuration of elements that exposes or enables particular dynamics.

To complement the textual description of their solution part, many patterns therefore include diagrams that illustrate and summarize this configuration of elements, the interactions within this configuration, and, if relevant, its evolution over time. These diagrams can help communicate the essence and detail of a pattern. They provide a graphical description of that pattern's 'big picture.' A diagram often says more than a thousand (and twenty-four)[10] words.

Diagrammability and Patterns

It has been suggested that the capability of providing such a diagram is a fundamental property of patterns. However, there are software concepts, such as very specific design and implementation decisions taken for very specific systems, for which it is also possible to provide an illustrating diagram, that are not patterns. A concept in software must fulfill many more properties before it can be called a pattern. The flip side of 'if you can't draw a diagram of it, it isn't a pattern' [Ale79] is that even if you can draw a diagram of it, it is not necessarily a pattern. The ability to draw a diagram may appear necessary for something to be a pattern, but it is certainly not sufficient.

A diagram may therefore be helpful to a pattern's readership, but we should be cautious in stating that diagrammability is a key identifying property of patterns. Humans are versatile and imaginative, which means that in practice it is possible to express any concept in human experience, no matter how abstract, through some form of diagram. It is true that a diagram of a particularly abstract concept may not convey its meaning effectively to all observers, but it is also true that not only is the diagrammability of something not sufficient for patternhood, it fails to distinguish a pattern from anything else that may be conceived, experienced, invented, or otherwise formed.

10. Trygve Reenskaug, ROOTS 2001 conference, Norway.

Perhaps a more useful distinction is to emphasize that the concept in question must be a designed artifact, as opposed to something occurring in nature, and that a diagram must be based on the design's spatial configuration. Of course this does not uniquely distinguish patterns from other design concepts, but as a way of thinking about diagramming it is perhaps a more useful path to take.

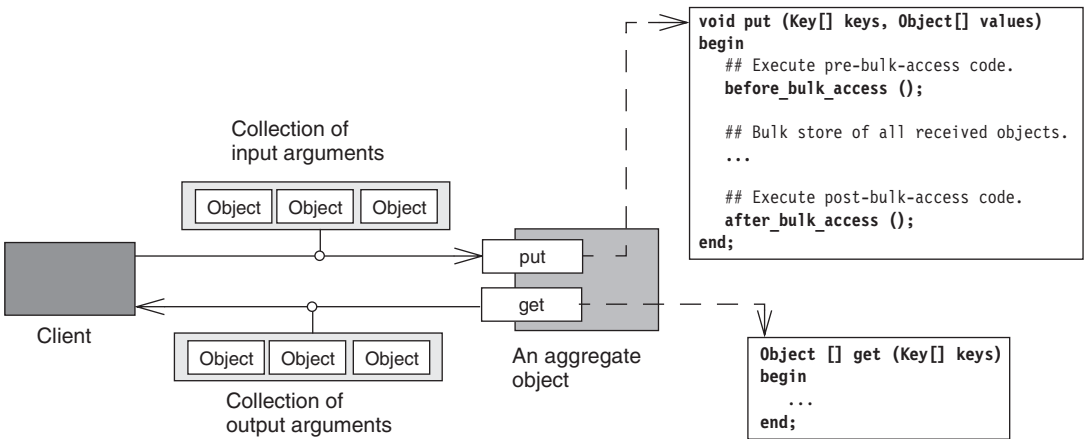
For example, it is certainly possible to create diagrams that illustrate fundamental design principles like 'separation of concerns' and 'encapsulation,' but such diagrams will tend to be general and abstract, and thus hard to communicate and discuss. When applied in a specific design, however, these principles come to life and are more tangible by developers. For example, our proto-pattern *encapsulates* the iteration over elements of an aggregate object within the data structure that is returned when accessing these elements:

> [...] Instead of accessing aggregate elements individually and directly from within a loop, the client invokes the method to access multiple elements in a single call. [...]

A diagram that illustrates this specific case of encapsulation is easier to understand, communicate, and discuss—and thus of more value in the context of a concrete design than a diagram that illustrates encapsulation in general.

Even the path of showing a concrete design, however, is not without its own pitfalls and pratfalls. The notion of space—and hence 'spatial'—in the aphysical and invisible domain of software is more about metaphor than matter [Hen03b]. It is a subtle but significant distinction, but one that should be kept in mind when carrying ideas from physical engineering and architectural disciplines to aphysical ones, such as software development. With the exception of software artifacts such as user interfaces, spatial concepts in software design derive from constructed visualizations that are the result of choice, as opposed to being a given. Consequently, a different choice of mapping can create a different notion of space. It is possible, therefore, to make a poor design look good through simple cosmetics and a flattering choice of abstractions, and a good design look poor by not paying enough attention to the choice and detail of visualization.

In advocating diagrams for patterns, it may be more useful to consider diagramming a matter of taste, form, and presentation than as something deeper. With this sensibility in mind, we can provide the following diagram for our proto-pattern.



This diagram intentionally does not follow popular modeling notations such as UML [BRJ98]. The reason for this is simple: a formal (UML) structure diagram is too often interpreted as the *one and only* true solution for a given problem. In the context of patterns, however, it is not the only solution! A pattern is generic, it defines roles, not classes, and so there can be many ways to implement it. A formal diagram, on the other hand, can often depict only one particular of the many possible configurations and interactions of these roles, or only one of their many possible evolution paths.

In addition, the more formal a diagram is, the more tempting it is to implement the pattern as specified in the diagram, because it appears to represent a standardized reference solution that is reusable wholesale and without adaptation in every situation to which the pattern is applicable. The OBSERVER story from *Chapter 0* illustrates this misconception nicely. The less formal a pattern diagram is, in contrast, the more developers are forced to think how to implement the pattern in their own systems, or how to specify its implementation in the design notation they use. Thinking is the essence of design, not (rote) automation.

If developers think in roles, however, and interpret any diagram as just an illustration of one of many possible concrete solution structures for resolving a particular problem, the specific notation selected becomes less important. When thinking in roles it may even be beneficial to follow known notations to illustrate the solution, because everybody is familiar with them. In much of the *Pattern-Oriented Software Architecture* series we assume that readers are familiar with the basics of the pattern concept and know that pattern participants denote roles, at least after reading the introduction to patterns in *A System of Patterns* [POSA1].

It was in exercising this matter of choice, therefore, that OMT and UML diagrams were favored over informal sketches [POSA1] [POSA2] [POSA3]. But this choice was made in context, and does not mean the same decision applies in all cases. For example, in *A Pattern Language of Distributed Computing* [POSA4] a less formalized, made-up notation is used to illustrate pattern structure. This 'Bizarro'[11] notation is also used here to illustrate our pattern-in-progress.

1.8 Evocative Names Help Pattern Recollection

To be used successfully, a pattern description must at least provide a clear and sound solution to a recurring problem that is well-presented through prose and pictures, as discussed above. But if we want to use patterns in our designs and implementations effectively, we must also be able to identify and reference each individual pattern quickly and unambiguously. In addition, even if we do not have particular pattern descriptions or their structure diagrams to hand, we still need a way of talking about these patterns and the designs in which they occur. In other words, we must be able to remember a pattern, otherwise it

11. The chunky and blocky appearance is reminiscent of the cubic Bizarro world, Htrae, home of Bizarro, a character from DC Comics' Superman series. Bizarro logic is also slightly inverted and different, giving Bizarro modeling a distinctly different feel to UML: where UML documents models different aspects through a multitude of diagrams, Bizarro notation uses only a single diagram that integrates multiple aspects. Where UML is formalized, Bizarro notation is more ad hoc: where UML is a marketable skill, mastery of Bizarro notation confers no career advantage (except on Htrae).

cannot become a word in our design vocabulary. A pattern that cannot be remembered is less likely to be recalled in practice, regardless of its other qualities.

Every pattern therefore needs a name. This name should be evocative [MD97]. Ideally, if someone references a pattern by its name, anyone familiar with it should be able to recall it from that cue alone. This familiarity is not always easy to achieve. Cute and obtuse names that are meaningful only to a handful of people, such as the clique of the pattern writer, are not necessarily meaningful to others. Recalling that a pattern is a vehicle for communication, a poorly named pattern offers poor transportation. Patterns are most memorable if their names conjure up clear images that convey the essence of their solutions to the target audience [MD97].

A Grammatical Classification of Names

Grammatically, two 'types' of names are used commonly in pattern naming:

- Noun-phrase names describe the result created by a pattern. For example, ACTIVE OBJECT and COMMAND PROCESSOR. Noun-phrase names typically describe the solution structure of the pattern, and in some cases may explicitly enumerate the key roles, for example, MODEL-VIEW-CONTROLLER or FORWARDER-RECEIVER.
- Verb-phrase names are imperative, giving an instruction that describes how to achieve a pattern's desired solution state. For example, the ENGAGE CUSTOMERS organizational pattern and the INVOLVE EVERYONE pattern for organizational change are both examples of verb-phrase names, as is DON'T FLIP THE BOZO BIT, which is one of the more colorful cliches of effective software leadership and interpersonal dynamics within groups.

Noun-phrase names are more common than verb-phrase names, and are normally preferred. Noun phrases highlight the structural nature of the solution and can be used most easily as part of an ordinary sentence. In considering patterns as being both 'a process and a thing,' noun-phrase names emphasize the 'thing,' whereas verb-phrase names emphasize the 'process.'

Literal Versus Metaphorical Names

The style of a name is also important. In conjuring up images of a pattern's essence there is a continuum with two contrasting extremes—*literal names* and *metaphorical names*:

- Literal names are direct descriptions of patterns that use terminology in its primary sense, rather than in a metaphorical way. For example, EXPLICIT INTERFACE and ITERATOR are literal names.
- Metaphorical names create associations between a pattern and another concept, such as one from everyday life, with which readers are hopefully familiar. VISITOR, OBSERVER, and BROKER are examples of metaphorical names.

Given the abstract nature of software development, many terms are based on metaphors to begin with—sockets, files, windows, and so on—so from a different point of view, some literal names may be considered metaphorical. Many names are therefore part literal and part metaphorical, for example, RESOURCE LIFECYCLE MANAGER.

Which type of name works best is often subject to personal preference. For example, REPEATING METHOD would be a literal noun-phrase name for our proto-pattern and BOXCAR METHOD a metaphorical noun-phrase name. Both names capture the essence of the pattern's solution equally well. We favored a third name, BATCH METHOD, which has elements of both naming styles. We could have chosen a name that was both literal and a verb phrase, but it is often hard to phrase short and crisp instructions that capture a pattern's essence precisely. For example, EXPRESS EACH FORM OF REPEATED ACCESS AS A LOOP-ENCAPSULATING METHOD is an instruction that captures the fundamental idea of our proto-pattern, but it is far too verbose to be a handy and evocative pattern name. Inconvenience is also not a good property of a transport medium.

1.9 Patterns are Works in Progress

After a long and scenic journey, we have finally arrived at the following description of the proto-pattern:

Batch Method

In a distributed or concurrent object system, the client of an aggregate object may need to perform bulk actions on the aggregate. For example, the client may need to retrieve all elements in a collection that have specific properties. If the access to the aggregate is expensive, however, because it is remote from the client or shared between multiple threads, accessing it separately for each element over a loop—whether by index, by key, or by ITERATOR—can incur severe performance penalties, such as round-trip time, blocking delay, or context-switching overhead.

How can bulk accesses on an aggregate object be performed efficiently and without interruption if access is costly and subject to failure?

Four forces must be considered when resolving this problem:

- An aggregate object shared between clients in a concurrent environment, whether multithreaded locally or distributed across a network, is capable of encapsulating synchronization for individual method calls but not for multiple calls, such as a call repeated by a loop.
- The overhead of blocking, synchronization, and thread management must be added to the costs for each access across threads or processes. Similarly, any other per-call housekeeping code, such as authorization, can further reduce performance.
- Where an aggregate object is remote from its client, each access incurs further latency and jitter and decreases available network bandwidth.
- Distributed systems are subject to partial failure, in which a client may still be live after a server has died. Remote access during iteration introduces a potential point of failure for each loop execution, which exposes the client to the problem of dealing with the consequences of partial traversal, such as an incomplete snapshot of state or an incompletely applied set of updates.

> For a given action, therefore, define a single method that performs the action on the aggregate repeatedly.
>
> Each BATCH METHOD is defined as part of the interface of the aggregate object, either directly as part of the EXPLICIT INTERFACE exported for the whole aggregate type, or as part of a narrower, mix-in EXPLICIT INTERFACE that only defines the capability for invoking the repeated action. The BATCH METHOD is declared to take all the arguments for each execution of the action, for example via an array or a collection, and to return results by similar means. Instead of accessing aggregate elements individually and directly from within a loop, the client invokes the BATCH METHOD to access multiple elements in a single call. The BATCH METHOD may be implemented directly by the aggregate object's underlying class, or indirectly via an OBJECT ADAPTER, leaving the aggregate object's class unaffected.
>
> A BATCH METHOD folds repetition into a data structure rather than a loop within the client, so that looping is performed before or after the method call, in preparation or follow-up. Consequently, the cost of access to the aggregate is reduced to a single access, or a few 'chunked' accesses. In distributed systems this 'compression' can significantly improve performance, incur fewer network errors, and save precious bandwidth.
>
> By using a BATCH METHOD, each access to the aggregate becomes more expensive, but the overall cost for bulk accesses has been reduced. Such accesses can also be synchronized as appropriate within the method call. Each call can be made effectively transactional, either succeeding or failing completely, but never partially.
>
> The trade-off in complexity is that a BATCH METHOD performs significantly more housekeeping to set up and work with the results of the call, and requires more intermediate data structures for passing arguments and receiving results. The higher the costs for networking, concurrency, and other per-call housekeeping, however, the more affordable this overhead becomes.

There is a marked difference between this version to the one with which we started: all the features that make up a good pattern description are now present. We have an evocative pattern name, a concrete and precise context, a crisply phrased problem statement, an explicit description of all forces that inform any viable solution, and an appropriate solution that resolves the problem and its forces well.

The latest solution description consists of a specific role-based structure that includes static and dynamic considerations, as appropriate, and a process to create this structure. It is also possible to draw one or more illustrative diagrams. The solution is concrete but also generic: it can be implemented in multiple ways that still preserve its essence and are recognizably the same pattern. Both the original and the final versions meet the 'a pattern is a solution to a problem that arises within a specific context' definition, but there are worlds between them. The latter can be considered a good pattern description, the former cannot.

Just as most useful software evolves over time as it matures, many useful pattern descriptions evolve over time as they mature. This maturity results primarily from the deeper experience gained when applying patterns in new and interesting ways. For example, the ABSTRACT FACTORY pattern evolved from the version in the Gang-of-Four book [GoF95], which allows only object creation, to a version that offers both object creation and object disposal [Hen02b] [Bus03a]. The later versions balance the aspects and forces of object lifetime management better than the original version.

Similarly, the original description of the BROKER pattern in *A System of Patterns* [POSA1] has been revised three times. The first revision was in *Remoting Patterns* [VKZ04], which decomposed the broker role of the original POSA1 version into several smaller, more specialized roles. This version was then revised a second time to use the original POSA pattern format and elaborate the implementation details of the revised structure [KVSJ04]. The third revision of BROKER, which is described in *A Pattern Language for Distributed Computing* [POSA4], extends the second and third version by integrating it with even more patterns that help implement BROKER-based communication middleware. All three revisions reflect a better understanding of the pattern itself, as well as its integration with the growing number of patterns that can be combined into a pattern sequence or language to implement it. But three need not be the number of the counting: as John Vlissides observes, 'The deliberations herein should convince you, in case you need convincing, that a pattern is never, *ever* finished.'

Sisyphus the Pattern Writer

We could try to improve our BATCH METHOD pattern further. For example, we could revise the context to cover even more situations in which the pattern applies, such as dealing with access to local, in-memory complex data structures (a situation that we discussed as an alternative context on *page 44*, but did not integrate into the pattern's context description). The fact that the solution uses object orientation could be expressed as a force rather than as a prerequisite in the pattern's context. We could also extend its description with known uses, such as examples of production software systems that have used this pattern successfully, and comparisons with related patterns, such as ITERATOR.

In other words, improving a pattern is an open-ended process. The pattern community therefore considers every pattern as a *work in progress*, subject to continuous revision, enhancement, refinement, completion, and sometimes even complete rewriting. Only this perspective allows a pattern to evolve and mature, as the various versions of our example pattern demonstrate. Just compare the patterns from the *Pattern-Oriented Software Architecture* series with their early versions published in the PLoPD series to see how they evolved over time.

Unfortunately, considering patterns as works in progress is a time-intensive process that demands a great amount of effort. This is one reason why there are many more pattern users than pattern authors, and why so many patterns do not escape their place of origin. On the other hand, the return on investment of this time and effort is the reward of the feedback that you receive from the software community and your own increased understanding of the patterns.

For example, if you take the time to discover and document useful patterns, developers may choose to employ the patterns you describe into new systems, or they may recognize them in existing systems. Spreading the word on specific good practices raises the level of design. Diving into the patterns to document them can only increase your knowledge, such that casual familiarity with a design solution is replaced by deep understanding.

1.10 A Pattern Tells a Story and Initiates a Dialog

Despite the fact that the 'final' version of our pattern is still a work in progress, the improvement is sufficient—and the length still short enough—that it does not need explicit section headings to guide the reader. The pattern still reads well enough that readers are carried naturally from one logical part to the next. This progression is another property of a good pattern: *it tells a story*—albeit a short one. More precisely, in the context that most interests us, it is a 'successful software engineering story,' to borrow an observation from Erich Gamma.

Bob Hanmer [CoHa97] takes Erich's observation further, describing how the pattern's name is the story's title, the context is the story's setting, the problem statement is its theme, the forces develop a conflict that is hard to resolve, and the solution is the story's catharsis: the new resulting context and the situation with the solution in place is the concluding 'and they all lived happily ever after.' As we will see in the rest of this book, however, there is often the prospect of a sequel.

A pattern, however, does not just tell 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 invites its readers to reflect on the problem being presented: its nature, the factors to be considered when resolving it, its various solutions, and which solutions are most feasible within the readers' own context.

On first reading, a pattern encourages people to think first and then to decide and act, explicitly and consciously, rather than blindly follow a set of predefined instructions. They receive guidance, but all activities they perform are under their own control. This difference allows the pattern to become part of the readers' design knowledge: over time experience becomes expertise that is applied intuitively rather than dogmatically.

1.11 A Pattern Celebrates Human Intelligence

Although the seeds of a solution may be found in a careful statement of the problem, the transition and transformation from one to the other is not always trivial or direct. 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 or model-driven tools, patterns are founded in human ingenuity and experience. Although a pattern's forces constrain the set of possible, viable solutions, these constraints are not so rigid as to ensure only a single outcome that can be captured as an automated transformation. In contrast, the constraints that bind a refactoring are strict and easily formalized: a refactoring alters the structure of code in a limited way so as to preserve its functional behavior.

In our example pattern the avoidance of the common ITERATOR pattern or even a humble subscripting index to perform iteration may be considered odd by many developers accustomed to thinking of these as the ordinary means for expressing iteration over an aggregate object's contents. ITERATOR has become the commonplace means for expressing iteration decoupled from a collection's representation in many mainstream languages, so much so that it has almost gone from being 'pattern' to 'default.'

What interrupts this tidy, uniform view of iteration design is the context of distribution, where the principle of minimizing the number of remote calls disturbs the peace. Little tweaks to the basic ITERATOR model do not work and a quite different solution is needed, one based on another line of reasoning.

1.12 From a Problem–Solution Statement to a Pattern

It is now easier to see that a pattern is much more than just a solution to a problem that arises within a specific context. On the other hand, this does not mean that this context–problem–solution triad is inappropriate for capturing patterns. It is an important form for describing patterns succinctly, as well as a denotation for every pattern's main structural property. It does not, however, specify how to distinguish a true pattern from an 'ordinary' solution to a problem. The context–problem–solution trichotomy is necessary for a specific concept to be a pattern, but it is not sufficient.

In case the intent of this chapter is misread, it is also worth clarifying that just improving a description does not automatically convert any given solution to a problem into a pattern. Patterns cannot be wordsmithed into existence. Only if a problem–solution pair *has* the other properties we discussed above can it be considered a good or whole pattern. The original example was just poorly expressed, but all the players were there or waiting in the wings, so stepwise refinement and piecemeal addition was possible. If a specific solution to a problem is lacking any of a true pattern's anatomical necessities, it is probably just *a* solution to a problem, and most probably a specific design and implementation decision for a specific system—but not a pattern.