1 Inside Patterns

A pattern is more than a solution to a problem within a context.

Now that the initial rave about patterns is over, it is time to revisit and reflect upon the notion of patterns itself. Not to create yet another pattern definition, or to correct existing ones. Instead we present and discuss the insights about patterns that we have gained over the past five years, after publishing *A System of Patterns*. and *Patterns for Concurrent and Distributed Objects*. These insights complement existing pattern definitions and thus help with understanding patterns at a deeper level.

1.1 Patterns: A Success Story

Patterns have taken the software development community by storm. Developers have been enthusiastic about patterns ever since the seminal work by the Gang-of-Four [GHJV95]. Its successors, which include collections of selected patterns from the Pattern Languages of Programming (PLoP) conferences [PLoPD1], [PLoPD2], [PLoPD3], [PLoPD4], and the first two volumes from the Pattern-Oriented Software Architecture series, *A System of Patterns* [POSA1] and *Patterns for Concurrent and Distributed Objects* [POSA2], further fanned the burning interest in patterns kindled by the Gang-of-Four book.

Software developers from all over the world jumped upon the 'new idea'—hoping that patterns would help them to resolve the one or other tricky problem with which they had struggled in the past. Patterns were applied in many development projects to get better designs and implementations. A movement started. It was, and still is, thriving. Today, there are many success stories to tell. Stories about systems which architectures took incredible benefit from being designed with patterns: telecommunication and business information systems [Ris98], industrial automation [BGHS98], and also CORBA compliant Object Request Brokers [POSA4].

The major reason for the success of patterns is that they constitute a 'grass roots' effort to build on the collective experience of skilled designers and software engineers. Software engineers who work on a particular design problem rarely invent a new solution that is completely distinct from existing ones. Instead they often recall a similar problem they once have solved in a different situation, and if its solution has proven to work, reuse the essence of this solution to resolve the new problem. Experts in software engineering know a large body of such solution schemes for common design problems from practical experience and follow them when developing new applications.

Distilling out commonalities in both application-specific design problems and their solutions then leads to the concept of patterns: they capture these solutions in an easily-available form. Patterns help novices to act as if they were—or almost as if they were—experts on modest-sized projects, without having to gain many years of experience. Patterns support experts in the design of large-scale and complex software systems with defined properties and also enable them to learn from the experience of other experts. No other concept in software engineering has similar or comparable properties.

1.2 Beyond Pattern Definitions

Almost all of the recent books on software engineering techniques include a definition of patterns for software, and if it is not the authors' own, they explicitly discuss or reference a definition coined by somebody else. The objective of these book authors is to introduce the concept of patterns, capture it as precisely as possible, and prevent its misinterpretation. The following is the definition we have developed in [POSA1]:

A *pattern for software architecture* describes a particular recurring design problem that arises in specific design contexts, and presents a well-proven generic scheme for its solution. The solution scheme is specified by describing its constituent components, their responsibilities and relationships, and the ways in which they collaborate.

Along with this definition we have presented and discussed several characteristics of patterns for software architecture. Every pattern

- documents existing, well-proven design experience,
- identifies and specifies abstractions that are above the level of single classes and instances, or of components,
- provides a common vocabulary and understanding for design principles,
- is a means of documenting software architectures,
- supports the construction of software with defined properties,
- helps with building complex and heterogeneous software architectures, and
- helps to manage software complexity.

Most other well-known definitions of patterns for software can also be boiled down to this characterization, such as the definition given by the Gang-of-Four [GHJV95] and those listed in [App96]. This is where we, and many others from the pattern community, were in 1996.

The definitions served well for introducing the pattern concept and are still useful today. Yet, people want to learn more about patterns. Once they understand the basic idea and applied patterns successfully in their own projects, they want to, for example, write their own patterns, or to learn more about the underlying philosophy. Now that the initial rave about patterns is over, it is time to serve that demand.

One way to do this is to re-write the existing definitions and enhance them with 'fine points' about patterns that go beyond their obvious properties. However, this is not our intention. We do not think it is a useful idea to come up with yet another pattern definition. Rather we like to contribute to the ongoing discussion about the characteristics of patterns. We do this by reflecting about what we have learned over time when using, teaching, and writing patterns in real-world practise. These 'ruminations' thus capture our experiences with patterns over the past five years. It is a set of stories about patterns and particular aspects of this concept.

1.3 A Solution To A Problem And More

If asked about a one sentence characterization of what a pattern is, most software engineers will likely respond:

'A pattern is a solution to a problem in a context.'

This seems to be—at a first glance—a fair summary. In fact, it is not false, as you can see by looking at the high-quality software patterns being around these days: they all provide well-working, concrete solutions to recurring organizational, analysis, design, and programming problems that may arise in specific situations during system development.

However, a pattern is more than that! If the above summary is sufficient, the following would be a pattern.

> A system that supports customer-specific behavior.
>
> For a service that must vary in a system it is often possible to define an implementation skeleton that captures this service's core processing scheme. Different versions of the service only differ in specific aspects, such as guards in conditional statements or particular actions that are to be performed. How can dependencies of the application to such a service be minimized?
>
> Implement each variant of the service separately from other variants. Provide all implementations with a common interface. Clients of the service then only depend on this shared interface, not on a specific service implementation.

This is obviously a solution to a problem in a context. It is not a pattern, however. But what is missing then?

A Process and A Thing

When analyzing the example 'pattern' with respect to content and its quality, the first deficiency that attracts attention is the vagueness of its solution. This solution only describes the process we should follow when resolving the problem. It does not specify what structure to create. Thus, there are many implementations possible, all of which are a product of following this process. One valid implementation would be, for example, a class that includes all service variants as separate private methods with different method names, and a single public method that dispatches to the currently used implementation. All service variants are separated, and there is a common interface to clients. However, this implementation is certainly not very practical—there are many better solutions of the original problem known.

A real pattern, instead, describes both a *process and a thing*, with the thing to be created by the process [Ale79]. For software patterns, 'thing' means a particular design structure or code, including its intended behavior [POSA1]. Actually, this corresponds to what software engineers do when building software systems: they follow a particular process to build a specific system. Unfortunately, popular processes do not tell us what to build, they only suggest a general procedure for how to build systems. Patterns tell us both, at least for resolving a specific problem, which is a big advantage they have over these processes.

The 'thing' to be build is missing in the example's original solution. We thus revise the solution paragraph and come up with a second version, as follows:

> Implement each variant of the service as a separate class. Provide all classes with the same public interface, which is to be used by clients to access the service. Clients then only depend on this interface, not on a specific service implementation. Configure the system with a particular implementation of the service, and connect it to its clients.

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

Best of Breed

Now that we know what to build, we see that the structure which the 'pattern's' second solution version proposes lacks of quality itself. It recommends to encapsulate every possible variant of a service in a separate class. The problem statement, however, says the following:

> For a service that must vary in a system it is often possible to define an implementation skeleton that captures this service's core processing scheme. Different versions of the service only differ in specific aspects, such as guards in conditional statements or particular actions that are to be performed. How can dependencies of the application to such a service be minimized?

This statement does *not* say: all service implementations are completely distinct from each other. Rather it says that all variants of the service share a common core and only differ in specific aspects. If we implement the second version of the solution, however, we would re-implement this common core for every service variant. This does not only result in huge coding overhead. Maintenance becomes tedious and error-prone also. A single change in the service's general processing schema affects all service implementations, which must be modified and debugged explicitly and separately.

To be useful for practical application, a pattern must therefore not propose just some solution to the problem. Rather it must present a *high-quality solution* which resolves the problem most optimally.

Moreover, the solution must be proven [POSA1]. Patterns do not represent neat or artificially created ideas that might work, but concepts that have been applied successfully in the past, over and over again. The Gang-of-Four put it this way: ‘Patterns distill and provide a means to reuse the design knowledge gained by experienced practitioners’ [GHJV95]. Brian Foote once coined the following, more pictorial phrase: ’Patterns are an aggressive disregard of originality.’ Consequently, new ideas must first prove to work, often multiple times, before they can be called patterns.

We revise the solution once again, and provide a third version:

> Capture the general processing schema for the service in a separate class. Specify a public interface for this class that is to be used by its clients to access the service. Clients then only depend on this interface, not on a specific service implementation.
>
> Define hook methods [Pree95] for all aspects of the service that can vary. This abstracts from their specific implementations. Use the hook methods in the implementation of the service’s general processing schema, instead of hard-wiring varying aspects directly into its code. Thus, when the service’s general processing schema executes, it delegates the execution of variant aspects to the hook method implementations.
>
> Declare the hook methods as private functions of the service class. Implement the hook methods for a specific service variant by subclassing the service class. Configure the system with the subclass that provides the required hook method implementations and connect it to its clients.

This third version of the solution resolves the original problem much better, because it avoids the problems identified for the second solution version.

Forces: The Heart of Every Pattern

The above discussion reveals that the problem addressed by the example ‘pattern’ is not as easy to resolve as it might appear on a first look. It is not the pure problem by itself that is to be considered. There are also a number of requirements to the solution, for example that it should be maintenance-friendly. However, it is impossible to derive such requirements from the plain problem statement. Yet the problem's solution should address them, to gain the desired quality.

If, on the other hand, such requirements are added to the problem statement, they would become explicit, and the problem's solution could deal with them appropriately. The same argument holds for the desired properties the solution should provide. Requirements and desired properties have a significant impact on the solution that a pattern proposes. The pattern community calls these influencing factors *forces*, a term borrowed from Christopher Alexander's pattern work [Ale79].

Forces tell us why the problem that the pattern addresses is a hard problem; why it requires some really 'intelligent' solution. Forces help us understanding the problem with respect to what other aspects must be considered when resolving it. Forces are also the key for understanding why the problem's solution is as is, and not something different. We therefore add the following four forces to the original problem statement:

> Changes of the service's invariant processing schema should neither affect the implementation of its variant aspects, and vice versa, nor the implementations of clients using the service.
>
> The general processing schema of the service should not be polluted with variant-specific behavior.
>
> It may be necessary to change variant aspects independently.
>
> Multiple variant aspects may be semantically related or otherwise dependent on each other, for example, because they provide access to, or operate on, a common data structure.

A quick double-check of whether or not the third version of the solution fulfils the above forces reveals that it already satisfies the first two of them. The separation of invariant from variant behavior in separate classes, together with the organization of these classes in a hierarchy, provide what these two forces require. The fourth force is resolved as a by-product of this design. All hook method implementations of a specific service variant are encapsulated in one class. If some hook methods are related or dependent on each other, their implementations can operate on common data structures, and share sub-algorithms. This avoids unnecessary programming overhead that otherwise would arise. Obviously, the fourth force requires no 'extra' structure in the solution, thus it might be tempting to remove it from the list of forces.

Unfortunately, the third force is not yet addressed by the third version of the solution—at least if there are two or more independent aspects of a service that can vary. Changing a particular hook method implementation requires to test the implementations of the other hook methods within the modified class, to exclude side-effects of the modification. Adding a new variant of a particular hook method requires to implement a new class that includes implementations of all other hook methods as well. Even worse, it may be necessary to implement specific versions of hook methods multiple times, if they appear in multiple versions of the service. The same problems that arose with the general processing schema in the second solution version thus now show up for the service's variant aspects in the third version: coding and maintenance overhead, which does not support to change varying aspects independently. However, we noticed this deficiency only because we thought hard about the forces and making them explicit. We thus provide a fourth solution version:

> Capture the general processing schema for the service in a separate class. Specify a public interface for this class that is to be used by its clients to access the service. Clients then only depend on this interface, not on a specific service implementation.
>
> Define hook methods [Pree95] for all aspects of the service that can vary. This abstracts from their specific implementations. Group the hook methods into several abstractions. Each abstraction represents a particular domain-specific or platform-specific concept and includes the corresponding hook methods that are semantically related or dependent on each other. Implement every different variant of an abstraction in a separate class.
>
> Use the abstractions and their hook methods in the implementation of the service's general processing schema, instead of hard-wiring varying aspects directly into its code. Thus, when the service's general processing schema executes, it delegates the execution of variant aspects to the hook method implementations.
>
> Provide the service class with template parameters to allow its configuration with classes that implement the required variants of every abstraction, and thus for the hook methods they comprise.

Now all four forces are resolved, and the solution to the original problem gained quality again. Note how the consideration of the third force in the new solution version also had a side-effect on the fourth force. Rather than being resolved for free—as it was in the third version of the solution—it needs a special treatment now, by introducing independent abstractions. This underlines the importance of listing all forces explicitly, even if they seem to be superfluous at a first glance.

Unfortunately, it might not always be possible to resolve all forces completely. Forces may likely contradict each other, for example if they address efficiency and changeability. In such cases a solution must *balance* the forces, so that each is resolved sufficiently enough.

The Context: Part of a Pattern or Not?

Now that the forces are added to the problem statement, and the solution is adjusted accordingly, we can turn attention to the context part of the example 'pattern:'

> A system that supports customer-specific behavior.

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

However, the context plays an important role in a pattern: it defines the situation *when* a pattern might be applicable. The more precisely this situation is described, the less likely developers will apply the pattern inappropriately. Furthermore, a precise context makes it easier to understand that there is a problem, and also why there is a problem.

It is therefore helpful to adjust the current context of the example 'pattern' such that it describes only the particular situation that can lead to the problem for which the 'pattern' specifies a solution. The current problem statement already provides parts of this information in its first two sentences. We therefore merge these with the original context and also add information about the design activity and the corresponding application's use-case that lets the problem arise:

> Systems that exist in different variants must be prepared for customer-specific service adaptation. For example, it may be nec-

> essary to modify the behavior of application-services to fit into a specific customer's business process. Yet it is often possible to define an implementation skeleton that captures such a service's core processing scheme. Different versions of the service only differ in specific aspects, such as guards in conditional statements or particular actions that are to be performed.

This context is much more precise with respect to describing the situation in which the problem that is addressed by the 'pattern' can arise and in which it may be applicable. Developers can easily avoid the pattern's application in wrong situations, for example, if different variants of the services are completely different in their implementations. In this particular case the pattern's solution would be like 'shooting with guns at a mosquito.' A simpler solution would be more appropriate, for example the one that we specified when modifying the original solution for the first time (see page 16).

Moving information from the problem to the context statement requires to rephrase the remaining part of the problem description such that it becomes meaningful again.

> How can a variant service in an application be implemented effectively if all service variants share a common core?

As a side-effect of removing context information from the problem statement, the 'real' problem shines through more clearly. It is short and crisply phrased, and we understand—from the context discussion—that this is a problem.

Removing the generality of the original context at the benefit of being more precise has one drawback, unfortunately. There may be other situations in which the problem addressed by the example 'pattern' can arise and where the same solution helps resolving the problem, such as the following one:

> Systems that run in different environments and/or on multiple platforms must be prepared for portability. Dependent on the particular environment and platform, the implementations of infrastructural services, such as for connection establishment between remote components, differ. Yet it is often possible to define an implementation skeleton that captures such a service's core processing scheme. Different versions of the service only differ in platform-specific aspects, for example the IPC mechanism to be used for remote communication.

Knowing that there may be even more such situations[1] lets a problem arise: how can the context's completeness be ensured? As discussed above, an overly general context that summarizes all situations which let a particular problem arise may lead to inappropriate applications of a pattern. A very precise context, on the other hand, may prevent developers from applying a pattern in other situations where it is applicable.

One way to cope with this problem is to start with a context that describes the known situations in which a pattern is useful, and to continuously update this context whenever a new such situation is discovered. The context section would then look-and-feel similarly to the Applicability section of the Gang-of-Four pattern description form [GHJV95]. There, the Gang-of-Four listed all situations they knew about where a pattern can apply.

Another way to resolve this problem is to follow the approach taken in [POSA2]. There we narrow the focus of the contexts to the general theme of the book: concurrency, distribution, and networking. The contexts of all patterns address only situations related to these topics. In a separate chapter, we then discuss the patterns' applicability in other domains and situations. This approach works well if a collection of patterns is centered around a common theme: the context attached to a pattern is lean, readers easily see when the patterns can apply in a particular domain, but other situations in which the patterns may help are not forget.

Unfortunately, both approaches still do not resolve the context completeness problem. The chance to forget a situation in which a pattern is applicable is very likely. As a result, neither the very general contexts in [POSA1] and this section's example 'pattern,' nor the split contexts in [POSA2], nor the applicability section of the Gang-of-Four patterns [GHJV95] really work. On the other hand, the two latter approaches seem to be more practical than a general

1. An example for yet another situation where the 'pattern' applies is preparing a service for multi-threaded software architectures. Critical sections in a service implementation must then be protected from concurrent access, as shown by the Strategized Locking pattern [POSA2]. For different platforms or for different use cases of the service, the type of lock being most appropriate may vary. Yet we omit this case here. The focus of this discussion is on the fact that there can be multiple contexts for a pattern, and on how to deal with this fact, rather than on covering all situations in which the example 'pattern' may apply.

context. It is perhaps better to support the application of a pattern x in only few situations, but then correctly, rather than to allow the application of a pattern everywhere, even if it is not useful.

A completely different approach is to consider the context *not* as part of an individual pattern. As shown above, there are multiple contexts for the example 'pattern' possible, but there is only one problem and one solution statement. Some members of the software pattern community therefore argue that a 'real' pattern comprises only the problem and solution statements. A context is only useful and necessary when describing pattern languages, where the context specifies how a particular pattern is integrated into the language. In other words, the contexts define a network of patterns—the pattern language—but the patterns are independent of the network, thus they do not need a context. Consequently, the context completeness problem discussed above simply does not arise. If there are multiple pattern languages in which a pattern is useful or needed, each will provide its own context for the pattern.

It really depends on the perspective taken, whether or not it is useful to consider the context as part of a pattern. From the perspective of an individual pattern it is essential to know in which situations a pattern can be applied. The question of how a pattern is integrated with other patterns is useful, but of minor importance. Thus the context is probably part of a pattern, which, unfortunately, lets the context completeness problem arise. From a pattern language perspective it is useful to know how its constituent patterns connect. It is not necessary to know which other pattern languages also include a particular pattern. Thus, the context is only needed for defining the pattern language; it is not needed to properly describe a particular pattern.

This chapter is about individual patterns, not pattern languages, thus it takes the patterns-eye view and considers the context as part of a pattern. This view also corresponds to the majority of the software patterns, which are not organized in pattern languages. We thus revise the context section such that it captures the known situations which allow the application of the example 'pattern,' as follows:

> Systems that exist in different variants and run in different environments or on multiple platforms must be prepared for customer-specific service adaptation, as well as for portability. For

example, it may be necessary to modify the behavior of application-services to fit into a specific customer's business process. Or, dependent on the particular environment and platform, the implementations of infrastructural services, such as for connection establishment between remote components, differ. Yet it is often possible to define an implementation skeleton that captures such a varying application service's or infrastructural service's core processing scheme. Different versions of the service only differ in specific aspects, such as guards in conditional statements, particular actions that are to be performed, or low-level APIs and operating system mechanisms that are to be used.

Independent of the 'context question,' the 'pattern' gained quality once again. Developers can get a 'good picture' of the situations in which it may apply, even if the new context does not comprise all such situations.

Genericity

With the revised context, problem, and solution sections, the example 'pattern' looks much more like being a real pattern than the version with which we started—but we are still not done yet. Reading the solution section very carefully reveals that it is very specific with respect to using object technology and templates. This might not be our intention, however.

In general, patterns are independent of a particular implementation technology.[2] It is possible, for instance, to implement the example 'pattern' using objects, templates, or function pointers. Neither implementation violates its core solution principle.[3] Therefore, if

2. The exception from this rule are idioms, which are sometimes referred to as programming patterns. Idioms deal with the 'professional' use of specific programming languages to resolve problems which arise due to the absence or presence of particular language features. Examples include memory management [BN94] and parameterized types [Cope92] in C++, multi-threaded programming in Java [Lea99], and object creation in C++ and Smalltalk [GHJV95]. Idioms also cover other important programming aspects, for example performance-tuning [PLoPD2]. They are thus dependent on the technologies addressed and language features used in their solutions.

3. Even patterns that seem to be dependent on a specific implementation paradigm, for example object technology, are often not: Proxy [GHJV95], for example, loses only a small fraction of its elegance by giving up inheritance; Strategy [GHJV95] can be implemented in C by using function pointers instead of polymorphism and inheritance.

using a specific technology fits best for the situation in which a pattern can be applied, this must be stated explicitly. If not, the solution section should not be dependent on this technology, because this would overly restrict the pattern's implementation space.

Let us assume that our goal was to provide an object-oriented solution, but not to let the applicability of the solution depend on the availability of templates in the programming language we use for its implementation. The first aspect can be 'fixed' either by adding another force, if the use of object technology is a requirement for resolving the problem, or by extending the context, if using object technology is a constraint in our situation. We decide for the latter and change the first sentence of the context statement:

> Object-oriented systems that exist in different variants and run in different environments or on multiple platforms must be prepared for customer-specific service adaptation, as well as for portability.

This is probably not the best way to indicate that object technology is used for the solution, but at least it is made it explicit.

Removing the dependency of the solution to using templates requires a different action. As is, the 'pattern' appears to be more like the description of a system-specific design decision, implemented in a specific programming language, rather than a *generic* solution that can be implemented in multiple ways without ever being twice the same. Genericity is an important property of patterns. Even though it is necessary to specify the concrete structure that is to be created—remember, a pattern is a thing and the process to create this thing—it is undesirable to let the solution excluding any valid implementation of this structure. These include the template version, but there are several other well-working implementations possible, which are not template based. Here is the fifth version of the solution:

> Capture the general processing schema for the service in a separate class. Specify a public interface for this class that is to be used by its clients to access the service. Clients then only depend on this interface, not on a specific service implementation.
>
> Define hook methods [Pree95] for all aspects of the service that can vary. This abstracts from their specific implementations. Group the hook methods into several abstractions, according to the Wrapper Facade pattern [POSA2]. Each abstraction represents a particular domain-specific or platform-specific concept and in-

> cludes the corresponding hook methods that are semantically related or dependent on each other. Implement every different variant of an abstraction in a separate wrapper facade class.
>
> Use the abstractions and their hook methods in the implementation of the service's general processing schema, instead of hardwiring varying aspects directly into its code. Thus, when the service's general processing schema executes, it delegates the execution of variant aspects to the hook method implementations.
>
> Provide the service class with a mechanism to allow its configuration with wrapper facade classes that implement the required variants of every abstraction, and thus for the hook methods they comprise. One way to implement this mechanism is to organize the wrapper facades for every abstraction in a separate class hierarchy, according to the Strategy pattern [GHJV95], and to let the constructor of the service class accept specific concrete wrapper facade classes at run-time using polymorphism. Another way to implement this mechanism is to provide the service class with template parameters to allow its compile-time configuration with concrete wrapper facade classes.

The fifth solution description abstracts from the use of one specific mechanism for connecting the service class with the hook method implementations. This was achieved by outlining all *feasible* implementation options for configuring the service class. Theoretically, even further implementation options are possible, such as function pointers, but these do not comply to the object paradigm that was specified in the context as a constraint for using the 'pattern.' As a consequence of this change, developers now have the choice between several reasonable implementations of the pattern, and not just one, as before.

The second change made was to connect the 'pattern' with the Wrapper Facade [POSA2] and Strategy [GHJV95] patterns. These citations do not only show how the 'pattern' is integrated with these patterns, they also provide hints to the developers on how to implement particular aspects of the example 'pattern' properly. For example, the Wrapper Facade pattern discusses in depth how to find the right abstractions. Moreover, it covers aspects not addressed in the solution description of the example 'pattern,' such as the options for handling errors in the service class that may occur in the wrapper facade classes.

Unfortunately, we are not yet set. Even though the new version of the solution is more generic than the previous one, it is still too ‘static.’ Only the dependency to a particular programming language feature is removed, but the use of object technology is still overly restrictive. Just fixing the context, as done above, was not enough. Consider the first sentence of the solution section:

> Capture the general processing schema for the service in a separate class.

Even though the pattern implies to use object technology, as indicated in the context, it may not be feasible to implement the general processing schema as a class of its own. What if an application class should, or must, offer a variety of services to clients? Shall each varying service be implemented as a separate class, with a Facade [GHJV95] in front that summarizes all their interfaces and dispatches client requests to the corresponding service implementation classes? Certainly not: this would introduce an overly complex structure with an unnecessary indirection.

A generic solution does not introduce classes, or components. A generic solution introduces *roles* which particular components of the system must take to resolve the original problem well. A role defines a specific responsibility of a system, including its interaction with other roles. Although it is recommendable to separate the implementations of different roles, this does not mean that they must be encapsulated in separate components. A single component can take multiple roles. Roles can be assigned to existing components and—if necessary—they can be encapsulated in their own components. If roles are clearly separated within the implementation of a component, they can be handled and maintained as if they were implemented in their own components. If necessary, developers could introduce role-specific interfaces so that clients only see the roles they need.[4]

Roles are the key for a seamless and optimal integration of a pattern into an existing software architecture, and for combining multiple

4. This approach is also used by a popular component technology: COM [Box97]. Furthermore, it denotes a general way to allow role-specific views onto larger components, manifested by the Extension Interface pattern [POSA2].

patterns to larger-scale designs. Thus, we revise the solution once again and provide a sixth version:

> Capture the general processing schema for the service in a template method [Pree95]. Clients that want to access the service must use the template method's public interface. By this they become independent of a specific service implementation. Implement the template method as part of the class which is responsible for offering the service to the clients, potentially among other services.
>
> Use the abstractions and their hook methods in the implementation of the service's template method, [...] Thus, when the service's template method executes, it delegates the execution of variant aspects to the hook method implementations.
>
> [...]
>
> Provide the class that includes the service's template method with a mechanism to allow its configuration with wrapper facade classes that implement the required variants of every abstraction, and thus for the hook methods they comprise. [...]

Now developers can implement the template method for the service's general processing schema either as part of another class or as a class of its own. The first option is most suitable if the component that provides the service must provide other services as well. If, on the other hand, the service is the sole responsibility of the component, the implementation of the template method in its own class is the better option.

Sometimes, however, the proper resolution of the problem *requires* to implement a role in a separate component or class. The division of hook methods in multiple wrapper facade classes, for instance, illustrates such a situation. Only if the hook methods are 'objectified' it is possible to maintain and configure them as required by the pattern's forces. A pattern will therefore state explicitly which particular roles must be implemented in their own components. For all other roles it introduces, the pattern will not prescribe their implementation, in order to not restrict the solution's genericity unnecessary.

The sixth solution version still describes the structure that needs to be constructed in order to resolve the problem. However, with roles there are much more implementation choices of the pattern available. Roles thus contribute significantly to the genericity of a pattern; much more than a strict class or component approach can ever do.

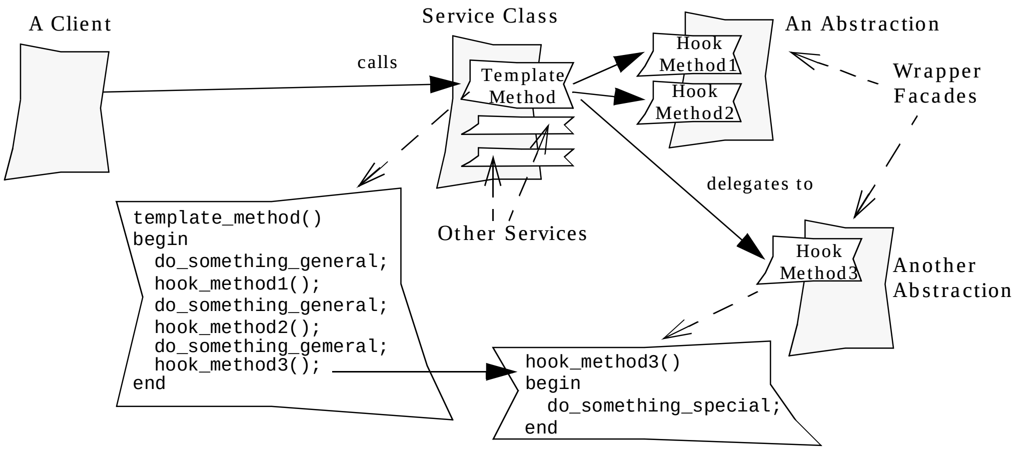
Roles also make it easier to adapt a pattern's core idea to the needs of a concrete application. Unnecessary complexity and indirection levels can be avoided, which leads to simpler, more flexible, and more efficient pattern implementations.

One Diagram Says More Than Thousand Words, Or Less

Now that the sixth solution version provides the quality we are expecting from a pattern, it is worth to spend some time on discussing a pattern's solution in general. Abstracting from its concrete look-and-feel, for most software patterns the solution specifies a design structure consisting of components, classes, and functions, that are connected through various relationships. These components, classes, and functions, represent the roles the pattern defines. The structure is completed by some behavior that 'happens' in the structure. In other words, the patterns specify abstractions that are above the level of single classes and instances, or of components [POSA1]. For idioms, the solution is a code fragment in which designated elements of a programming language are arranged in a specific way, together with the code's behavior. For process patterns, the solution defines a particular organizational structure, roles in this structure including their responsibilities, and the communication between these roles. Speaking most generally, a pattern defines a spatial configuration of elements that expose a particular dynamics.

To complement the textual description of the solution, many patterns therefore include a structure diagram that illustrates this spatial configuration of elements. Some members of the pattern community even argue that the capability to provide a structure diagram it is a fundamental property of patterns: if it is impossible to draw a structure diagram for a particular concept in software engineering, it is not a pattern. It might be a design principle, such as information hiding, or a step in a process or methodology, or any other concept for which it is impossible to draw a diagram that illustrates a particular spatial configuration of elements. Note, however, that 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 a structure diagram, but which are no patterns. A concept in software must fulfil many more properties before it can be called a pattern.

For the example 'pattern' we can provide the following diagram:'



```
template_method()
begin
  do_something_general;
  hook_method1();
  do_something_general;
  hook_method2();
  do_something_gemeral;
  hook_method3();
end
```

```
hook_method3()
begin
  do_something_special;
end
```

The above diagram is not following the notations of popular analysis and design methodologies, such as UML [BRJ98]. This is intentional and the rationale for this is simple: in the past a common misconception about patterns was to interpret their class or structure diagrams as *the* solution. However, it is not! A pattern is generic, and there can be multiple ways to implement it. A structure diagram can depict only one particular of the many possible configurations of elements that a pattern proposes. Unfortunately, the more 'formal' this diagram is, the more it is tempting to implement the pattern as specified in the diagram. Developers tend to 'forget' that a pattern introduces roles, because the diagram raises the impression that it is complete and thus 'reusable' in every application where the pattern is applicable. The less 'formal' the diagram is, on the other hand, the more developers are forced to think how to implement the pattern in their system, or how to specify its implementation in the design notation they use. If developers think in roles, however, and interpret any diagram as just an illustration of a pattern's solution, the kind of notation becomes less important. It may even be beneficial to follow known notations, because everybody is familiar with them. In the POSA series we assume that readers know the pattern concept, thus we have used OMT [RBPEL91] in [POSA1] and UML [BRJ98] in the subsequent volumes.

Pattern Forms: A Communication Vehicle

When patterns became popular, many people unified the notion of patterns with pattern description forms. Only software concepts that were documented in one of the known pattern forms were considered as a pattern, even if they were no patterns at all. Software concepts that were not documented ’right,’ were not accepted as patterns, even if they were. This is not true, however. A pattern, like any other concept, is independent of its presentation; the form is ’just’ a vehicle for communicating a pattern. Yet form is not unimportant, and patterns are probably the one concept in software technology, where form is an integral part of the whole idea. The intent of the pattern community is to spread the word about good practices in software development. If it is hard to communicate these good practices, however, it is as if they were not existing. In his essay ‘Writing Broadside,’ Richard Gabriel [Gab96] cites Thucydides, who—more than two millenniums ago—wrote:

‘*A man who has the knowledge but lacks the power to clearly express it is no better off than if he never had any ideas at all.*’ [Thu81]

Therefore, an appropriate description form is essential for communicating a pattern successfully, and for the success of a pattern its proper communication is essential. Unlike when listening to a talk, a reader of a pattern has not its author at hand, when certain aspects are unclear and must be explained, or when developers are stuck during a pattern’s implementation. The description by itself must express a pattern precisely, leaving no space for interpretations where it is not intended. Only then a particular pattern can become part of the common design vocabulary [GHJV95] [POSA1], so that developers know what problem and solution structure is meant when being told just a pattern’s name. Only then it is possible to talk about how to use a particular pattern in a specific system effectively, rather than discussing endlessly about a specific design principle.

There are a lot of different forms for documenting patterns from which we can chose: the very structured and detailed Gang-of-Four and POSA forms, the short and essence-focused form introduced by Jim Coplien, Ward Cunningham’s more narrative Portland form, the form used by Christopher Alexander, and even ‘unusual’ forms, such as the one developed by Alistair Cockburn, which introduces aspects like the ‘overdose’ effect of a pattern. The example pattern from this section, could, for example, easily be transformed into the Coplien form by adding the appropriate headings to each part, and extending the pattern with a resulting context and rationale section. It could also be converted to the POSA form by adding the missing sections, diagrams, graphics, and CRC-cards. Or, it could stay as is, just as plain prose. In all these forms we still document a pattern. Just browse the available body of pattern literature, such as [AIS77], [PLoPD1], [PLoPD2], [PLoPD3], [PLoPD4], [GHJV95], [POSA1], [POSA2], [Bec97], [Fow97a], to get an idea of the variety of different forms.

Which form is the right for describing a particular pattern, however, cannot be generally defined. It depends very much on the intent a pattern author has in mind with the description, and also the target audience she wants to address. If a pattern description intends to address managers or project leaders, for example to raise awareness for a problem and summarize the way out, a short form, such as the Coplien form, is better than a form that discusses the very details of a pattern’s implementation. The description of the ’pattern’ we have developed in this section is a good example for this kind of pattern form. Short forms are also very helpful for browsing the existing pattern space to find patterns that might be helpful for resolving a particular problem. On the other hand, if a pattern author wants to guide developers in implementing a pattern, such a short form is insufficient: a description of a pattern’s structure and dynamics, implementation guidelines, and concrete examples are needed in addition. Many useful hints and suggestions about pattern forms can be found in Gerard Meszaros’s and Jim Doble’s ‘Pattern Language for Pattern Writing’ [PLoPD3].

Our goal for the *Pattern-Oriented Software Architecture* series, for example, is to help software developers designing and implementing software systems. Thus we need to capture both the essence and the details of a pattern. The description form we have defined in [POSA1] is perfectly adjusted to these needs: it allows us to draw the ‘big picture’ of a pattern, to detail its concrete structure and dynamics, and to guide the implementation of the pattern described. Readers, who want to grasp the key ideas behind a specific pattern, only need to read its Context, Problem and Solution sections. These three sections describe the 'real' pattern—all other sections 'only' provide additional information. The Structure, Dynamics, and Consequences sections give an overview about a pattern's look and feel. The Implementation, Example Resolved, Variants, Known Uses, and See Also sections, finally, provide the details for applying and implementing the pattern in a real-world application.

Form is not the only aspect that impacts the success of communicating a pattern, however. Writing style is very important too. If a pattern description is hard to read, passive and formal, deals with many aspects at the same time and introduces complex issues too early, or if it requires to be an expert in the subject to be understood, it does not attract attention, and discourages readers from reading further. A pattern description should be precise, informative, fun to read, and telling a story. Only then it can be communicated successfully. Useful hints for an effective pattern writing style can be found in Richard Gabriel's book 'Patterns of Software — Tales from the Software Community' [Gab96], and in 'Pattern Hatching — Design Patterns Applied,' the book by John Vlissides [Vlis98].

A Recap

After a long journey, we finally arrived at the following description of the example 'pattern:'

> Systems that exist in different variants and run in different environments or on multiple platforms must be prepared for customer-specific service adaptation, as well as for portability. For example, it may be necessary to modify the behavior of application-services to fit into a specific customer's business process. Or, dependent on the particular environment and platform, the implementations of infrastructural services, such as for connection establishment between remote components, differ. Yet it is often possible to define an implementation skeleton that captures such a varying application service's or infrastructural service's core processing scheme. Different versions of the service only differ in specific aspects, such as guards in conditional statements, particular actions that are to be performed, or low-level APIs and operating system mechanisms that are to be used.

How can a variant service in an application be implemented effectively if all service variants share a common core?

Four forces must be considered:

- Changes of the service's invariant processing schema should neither affect the implementation of its variant aspects, and vice versa, nor the implementations of clients using the service.
- The general processing schema of the service should not be polluted with variant-specific behavior.
- It may be necessary to change variant aspects independently.
- Multiple variant aspects may be semantically related or otherwise dependent on each other, for example, because they provide access to, or operate on, a common data structure.

Capture the general processing schema for the service in a template method [Pree95]. Clients that want to access the service must use the template method's public interface. By this they become independent of a specific service implementation. Implement the template method as part of the class which is responsible for offering the service to the clients, potentially among other services.

Define hook methods [Pree95] for all aspects of the service that can vary. This abstracts from their specific implementations. Group the hook methods into several abstractions, according to the Wrapper Facade pattern [POSA2]. Each abstraction represents a particular domain-specific or platform-specific concept and includes the corresponding hook methods that are semantically related or dependent on each other. Implement every different variant of an abstraction in a separate wrapper facade class.

Use the abstractions and their hook methods in the implementation of the service's template method, instead of hard-wiring varying aspects directly into its code. Thus, when the service's template method executes, it delegates the execution of variant aspects to the hook method implementations.

Provide the class that includes the service's template method with a mechanism to allow its configuration with wrapper facade classes that implement the required variants of every abstraction, and thus for the hook methods they comprise. One way to implement this mechanism is to organize the wrapper facades for every abstraction in a separate class hierarchy, according to the Strategy pattern [GHJV95], and to let the constructor of the service class accept specific concrete wrapper facade classes at run-time using polymorphism. Another way to implement this mechanism is to provide the service class with template parameters to allow its compile-time configuration with concrete wrapper facade classes.

The difference of this description to the description with which we started is evident. All quality aspects that make up a pattern are now present: there is a concrete and precise context, a sharply phrased problem statement, an explicit description of all forces that influence the solution, and a high-quality solution that resolves the problem and the forces well. The solution consists of a specific structure, including its behavior, and a process to create this structure, and it is possible to draw an illustrating structure diagram. The solution is very concrete, but still generic: it can be implemented in multiple ways without violating its essential core. Both the original and the new description meet the 'a pattern is a solution to a problem in a context' definition, but there are worlds between them. The one is a pattern, the other is not.

The description improved so much that it does not even need section headings. The pattern still reads well and readers are also able to distinguish its various parts. In fact, this is another property of a good pattern: *it tells a story*. And it is structured like a story. Bob Hanmer once came up with the following analogy [CoHa97]: the context of a pattern compares to a story's setting. The problem statement is its theme, and the forces develop a conflict that is almost impossible to resolve. The solution is the story's catharsis, the *Deus ex machina* in a morality play. The new resulting context, the situation with the solution in place, is like the 'they lived happily ever after' in fairy-tales.

The phrase 'Deus ex machina' from above, which means 'God out of a machine,' is not accidental. A solution that a pattern proposes cannot be derived from the problem statement easily and in a straight forward fashion. For example, by applying an analysis and design method. A 'pattern-worthy' solution tackles a problem in an unusual, non-direct way. This requires human intelligence and experience.

In the example pattern the unusual solution principle is the inversion of the control flow. Developers are unable to override the template method and thus the service's general processing schema. They just can provide implementations of customer-specific and platform-specific aspects of the service—implemented as hook methods that are called by the 'untouchable' template method. This principle is often referred to as the 'Hollywood Principle,' that is 'Don't call us, we'll call you.' The 'Hollywood Principle' differs from past practise in object technology. There the solution to the example problem likely consisted of an abstract class that only declared the service's interface, with the various service variants fully implemented as separate subclasses, similarly to the second solution version in this chapter.

You see, a pattern is much more than just a solution to a problem in a context. On the other hand, this does not mean that the context-problem-solution triad is insufficient or inappropriate for capturing patterns. It is both an important form for describing individual patterns and a denotation for such a pattern's main property. However, it does not specify how to distinguish a 'real' pattern from an 'ordinary' solution to a problem. In other words, the context-problem-solution dichotomy is necessary for a specific 'concept' to be a pattern, but not sufficient.[5]

It is also noteworthy to say that it is impossible to turn any given solution to a problem into a pattern—as this section might imply—just by improving its description. Patterns cannot be constructed! Only if a solution *has* the properties we have discussed it is a pattern. The original example was just badly expressed, for reasons of motivating the different issues, so that it was possible to step-wise mine the aspects that let it make a pattern. If a specific solution to a problem is lacking either one of the properties we have discussed, it is just *a* solution to a problem in a context, most likely a specific design and implementation decision for a specific system, or a design principle, but not a pattern.

5. Note that for patterns in a pattern language the context is not necessarily be considered as a part of a pattern, as we have briefly discussed in this chapter.

1.4 A Million Different Implementations

One of the most important statements in the previous section is:

'A patterns provides a generic solution that can be implemented in multiple ways without ever being twice the same.'

We have discussed in depth what this means from the pattern perspective itself. However, reflecting about the concept of patterns is just one face of the coin. Building software with patterns is the other. From this code-eye perspective, one way to interpret the above statement is as follows:

Every pattern implementation must consider requirements that are specific to the software system in which it is applied, so that each is likely to be different from other implementations of that pattern. It is also impossible to create these different implementations by configuring and adapting a common reference implementation.

We want to evaluate this interpretation by taking the opposite position: we assume that it is possible to provide configurable reference implementations for patterns, and try to develop one for a concrete example, the Observer pattern [GHJV95]. If we fail we have proven the hypothesis to be false, which in turn evaluates our original statement to be true.

Structural and Behavioral Variation

The first challenge that arises when trying to develop a reference implementation for a pattern is that patterns introduce roles, and not classes—as we have discussed above. In the Observer pattern, for example, it is possible to assign the role of the subject to either an existing class of the application under development or to a new class. The same holds for the observer role. In some situations it might even be most effective to combine both roles in a single class. Or, by implementing the Event Channel variant [POSA1], developers can completely decouple the subject from its observers. Either variant results in a different Observer structure. This already contradicts to the idea of a general reference implementation.

There are also behavioral aspects of Observer that can vary. For example, the implementation of the pattern's change propagation mechanism can follow either the push or the pull model to exchange data between the subject and its observers. A third option is to implement an event-specific notification mechanism. Such behavioral aspects can vary independently of the pattern's structure and are also implemented differently in different structural variants of the pattern. For example, the event-specific pull model is implemented differently in a 'traditional' Observer [GHJV95] structure than in the Event Channel variant [POSA1]. In the one case the subject notifies observers directly, in the other there is an indirection via the event channel component.

However, even when focusing on a very specific structural and behavioral variant of a pattern it is still hard to provide a reference implementation for it. In the Observer pattern, for example, the subject usually uses a container to maintain the references it keeps to all registered observer objects. For the container's implementation many options are possible. Some kinds of containers are easy to implement, others are very efficient—due to the different data structures they use. Which of these is the most optimal one, however, cannot be decided independently of the context set by an application that is using the Observer pattern. Another application-dependent aspect is error handling. What happens if, for some reason, an observer that registers with the subject cannot be inserted into the container. For example, because the system runs out of memory? Is the container responsible for dealing with this situation, the subject, or the observer that tries to register?

A similar discussion can be hold for the Iterators [GHJV95] which are often used to traverse over the observer identifications maintained in the container. Should they be robust or not? What happens if an observer that is currently referenced by an iterator is unregistered from the subject and then deleted? The answer to this question can again not be given in advance. It is the application-specific requirements to the implementation of Observer that finally determine which option is the best.

This discussion already reveals that two implementations of a pattern will likely vary, but it does not yet exclude the configurable reference implementations—at least not if these are restricted to particular structural and behavioral variants. We could, for example, provide an Observer implementation that can be configured with specific kinds of containers and iterators.

Domain-Dependent Variation

There is one strong argument against such configurable reference implementations, however: every pattern implements behavior which is completely dependent on the application domain of the system in which the pattern is used. In the Observer pattern, for example, these are all state modifying methods in the subject which must trigger the change propagation mechanism, and all methods within the observers which must react on a change notification from the subject. On the subject-side developers must decide when to start the change propagation, to avoid unnecessary updates or update cascades [GHJV95]. On the observer-side they must decide whether a change notification should always result in updating data, or if just specific ones start the update, for example every N^{th} notification. A subject could also let a certain period of time elapse after every update before triggering the next [GHJV95].

How to implement all these aspects only depends on the application functionality which must be coordinated with help of Observer. An implementation that is suitable for one application is very likely unsuitable for others.

To further complicate matters, a pattern implementation must also fulfill general requirements to the application, for instance real-time and security aspects, or run-time configurability. Moreover, the implementation is influenced by decisions made earlier in the system's design process. For example, the communication between subjects and observers is to be implemented differently if they are located in the same thread of control, or in different threads, or even in different address spaces. It might also be necessary to use a specific communication platform to implement the collaboration, such as CORBA [OMG98] or DCOM [Box97].

Consequently, a reference implementation for Observer will quickly degenerate to just a set of interfaces which prescribe some essential methods, like `register()`, `unregister()`, `notify()` and `update()`, and their parameters. This is, however, not what is commonly understood as being a reference implementation. In addition, such interfaces are more a burden for developers rather than a help. Not only must they implement most parts of the patterns by themselves, they are also forced to use the 'reference interfaces' even if they are unsuitable for their application. In addition, these interfaces are probably declared in their own classes, if they are part of a library. Application classes that want to use the interfaces must therefore inherit from these library classes—just assigning the subject or observer role to an application class, as discussed in the previous section, is impossible.

Conclusion

As a result of this discussion, the vision of reference implementations for patterns is simply a myth. Patterns are no blueprints! Alexander is right when he says [AIS77]: 'Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it twice the same.'

In other words, a pattern's implementation for a specific application is rarely reusable, because it is tuned to this application's specific requirements.[6] It is also next to impossible to use a specific implementation of a pattern more than once in a software system, because the functional aspects to be considered by each of its uses are different.

Part of the power of patterns stems from the fact that they actually do *not* prescribe a particular implementation; that they instead capture the commonalities which many specific solutions of a single problem share, and that they support adapting this common core to a new specific situation inside another system. A pattern describes a whole *design space*. Concrete implementations are carved by the applications in which it is applied, because it is these applications' very specific requirements which define the path to be taken through the design space that the pattern provides.

6. This corresponds to the reusability of analysis and design models. They may be reusable in other applications than they were originally designed for, but their implementation is not. Similarly, a pattern is reusable, but not its implementation.

Patterns Versus Frameworks

People might argue, however, that there are already pre-fabricated implementations of patterns available which are widely and successfully used. Examples include the Smalltalk libraries [KP88], ET++ [Gam91], or ACE [Sch97]. This is by no doubts true. On the other hand, these libraries are *not* offering general purpose reference implementations of particular patterns. They are frameworks, which provide a pre-defined architecture and partial implementation for a particular family of applications. The Smalltalk library and ET++, for example, support the construction of systems with a graphical user interface. ACE supports developing distributed and concurrent applications. All frameworks take advantage of patterns, but implement them in a very specific context and for a very specific purpose. Only if a system obeys to such a framework's logic—because it is a member of the application family this framework represents—it is possible to use their pattern implementations successfully. If not, it is hard to use them in a meaningful manner.

It is very important to note at this point that it is no disadvantage to provide pre-fabricated pattern implementations within frameworks. In fact, an important reason why frameworks can speed-up software development significantly is that they implement patterns. An application that belongs to the system family represented by the framework can benefit from its pattern implementations by completing them with the application-specific logic. On the other hand, such an application does not leave the context set by the framework. Implementing a framework is thus different than calling for a general purpose reference implementation of a pattern that can be integrated into every application.

Patterns Versus Formalisms

A very similar discussion can be hold on formalizing patterns. There are many people arguing that a plain textual description of a pattern illustrated with some diagrams is not precise enough to ensure its correct implementation.

However, due to the many structural, behavioral and programming options, and the strong dependency on application-specific requirements and constraints, only few aspects of a pattern can be formally specified if the pattern's genericity should be preserved. Otherwise the specifications tend to describe only one particular of all possible pattern implementations, or just very few. Such a rigorous restriction, however, does not meet the needs of most applications that want to use the specified patterns. On the other hand, if many important parts of the solution are left unspecified, developers who want to use such specifications must define all missing aspects on their own. Furthermore, they also have to learn the formalism, and must deal with specifications which often enough are longer than the complete code they finally will produce. Again this is more a burden than a help.

Consequently, the genericity of a pattern's solution conflicts with the necessary level of detail required by formal approaches. Note, that this does *not* mean that it is useless to formally document *specific* pattern implementations. For example, formal techniques, such as design by contract, help to prevent an incorrect use of a pattern implementation within a particular application. Other formalisms, such Aspect-Oriented Programming (AOP) [KLM+97], allow to specify particular pattern implementations more precisely than informal design diagrams can ever do. However, a formal description of a very specific pattern implementation is different from trying to formally capture all possible implementations of a pattern.

1.5 No Pattern is an Island

Many pattern authors and experienced pattern users note that no pattern exists in isolation; that they rather expose manifold relationships and interdependencies among each other—in general and even more when being applied in real-world applications.

The Gang-of-Four book [GHJV95], for example, includes a map that illustrates the dependencies among the 23 patterns it presents. Furthermore, the 'Related Patterns' sections of the Gang-of-Four pattern descriptions provide concrete information about how the pattern being described is related to other patterns.

In [POSA1] we have discussed that a pattern may refine other patterns, be a variant of another pattern, or combines with other patterns. The pattern descriptions in [POSA1] and [POSA2] thus reference all other patterns that are related in one of the above ways to the pattern being described. For example, the description of the Model-View-Controller pattern [POSA1] refers to eight other patterns: Presentation-Abstraction-Control [POSA1], Publisher-Subscriber [POSA1], Command Processor [POSA1], Factory Method [GHJV95], View Handler [POSA1], Composite [GHJV95], Chain of Responsibility [GHJV95], and Bridge [GHJV95]. The description of the pattern developed in Section 1.3, *A Solution To A Problem And More*, includes references to Wrapper Facade [POSA2] and Strategy [GHJV95], as well as the discussion on template methods and hook methods in [Pree95].

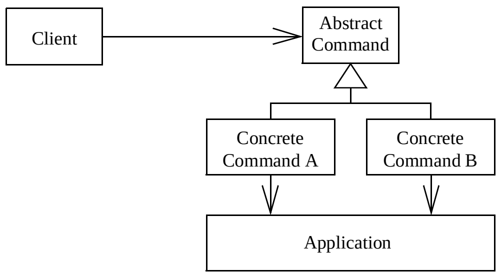
Pattern languages connect their constituent patterns even stronger. Every pattern in a pattern language may precede other patterns in the language, or complete other patterns, or both. Every pattern also describes how and in which order to apply the patterns that help implementing the solution of the problem it addresses. Thus, patterns of a pattern language can rarely 'live' outside the language, because they are tightly coupled and make only sense in the presence of the others.

The manifold concrete relationships and interdependencies that can exist between patterns are widely discussed in the body of pattern literature and also throughout this book. In this section we therefore want to focus on *why* the relationships and interdependencies between patterns are of significant importance for their successful application in real-world systems.

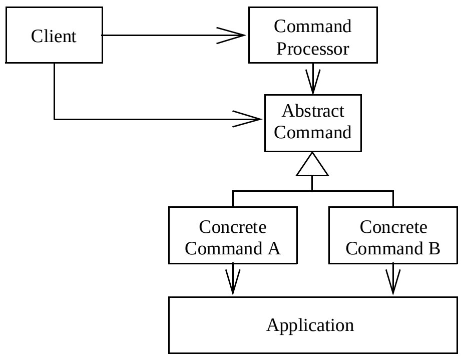
An Experiment: Patterns as Islands

To illustrate this, we assume that no relationships between any two patterns exist, and that no integration is allowed in their implementations. In other words, we see every pattern as an island. Under this pre-condition we want to use patterns in the development of an extensible request handling mechanism.

The first design problem to resolve when designing the request handling mechanism is to 'objectify' concrete requests, which clients, whether being human users of the system or other systems, can issue. The Command pattern [GHJV95] from the Gang-of-Four helps with this step. The initial design thus provides a class AbstractCommand which offers methods for executing client requests. Concrete requests understood by the application are implemented as separate ConcreteCommand classes, which derive from AbstractCommand and implement its interface. When clients of the system issue a specific request they instantiate a corresponding ConcreteCommand object and call its execution interface. This object then performs the requested operations on the application and returns the results, if any, to the client.



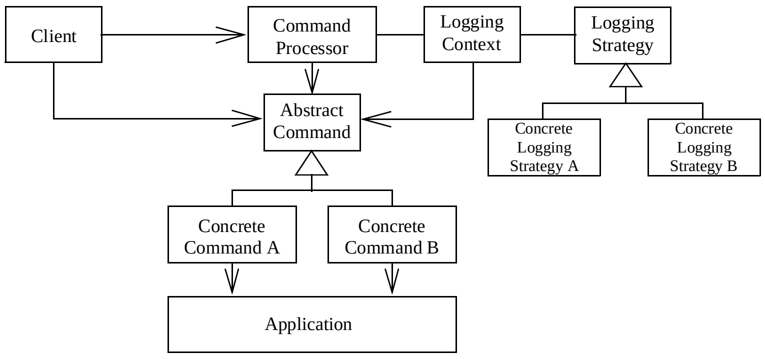
There may be multiple clients of the system that can issue commands independently. Thus it is useful to co-ordinate the general command handling within a central component. The Command Processor pattern [POSA1] provides such a structure. It introduces a CommandProcessor to which clients pass the commands they create for further handling and execution. Integrating the CommandProcessor class with the existing design is straight forward: it is inserted between the clients and the AbstractCommand class.



We further assume that it is necessary to provide logging functionality for requests. An aspect to consider in this context is that different customers of the system may want to log requests differently: some will log every request, others just particular kinds of requests.

The candidate pattern for resolving this problem is Strategy [GHJV95]: it supports the encapsulation and exchange of algorithms that can vary in a system. The integration of Strategy into the design, however, lets a problem arise, due to the rule that is not allowed to integrate patterns with each other. From a pragmatic perspective it is in the responsibility of the `CommandProcessor` to provide the logging service. However, the logging service's invariant parts are to be implemented in the context component that Strategy introduces.

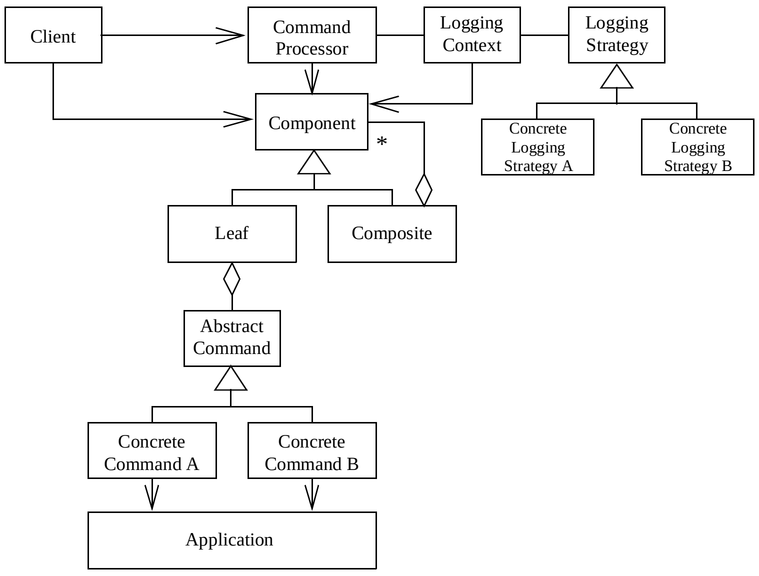
Strategy is therefore applied as follows: The `CommandProcessor` passes the `ConcreteCommands` it receives to a `LoggingContext` object which is a participant of the Strategy pattern. This object implements the logging service, but delegates the computation of customer-specific logging aspects to a `ConcreteLoggingStrategy` object. A class `LoggingStrategy` offers a common interface for all `ConcreteLoggingStrategy` classes, so that they can be exchanged without major modifications to the `LoggingContext` class. This is not the most elegant application of Strategy, but it satisfies the rule for not integrating patterns.



A final step is to support macro commands. A `ConcreteCommand` may be an aggregate for other `ConcreteCommands` which are to be executed in a specific order. The design pattern that provides such a structure is Composite [GHJV95]: macro commands could be represented as composite objects and 'atomic' commands as leaf objects. However, due to the prohibition of integrating patterns, adding Composite to the design is not an easy task: clients cannot create `ConcreteCommands` directly.

Instead they must instantiate `Component` objects, which may either be `Composite` or `Leaf` objects, according to the Composite pattern. `Leaf` objects include a reference to an `AbstractCommand` and thus can represent a specific `ConcreteCommand`. `Composite` objects in turn can aggregate several `Leaf` objects and therefore several `ConcreteCommand` objects, which then form the macro command.

The final architecture of the request handling mechanism looks very complex, and in fact it is. The design includes four patterns, but the it does not expose the expected quality: it is hard to understand, maintain, and extend. Moreover, due to the many components being involved in executing a request, the implementation of this design is very inefficient.

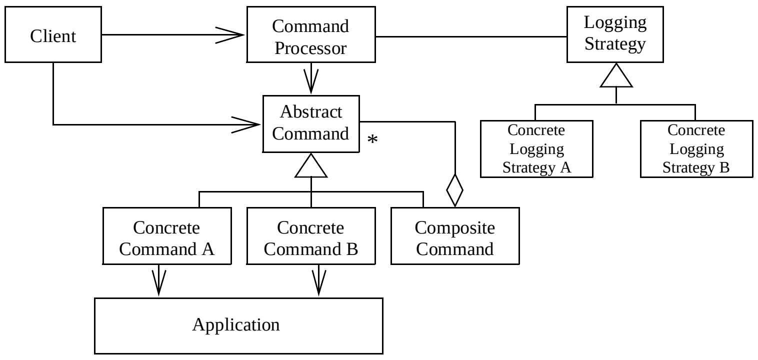


A Second Experiment: Interwoven Patterns

If we are allowed to combine and integrate the patterns with each other, however, the design of the request handling mechanism will look and feel completely different. For example, it is possible to assign the responsibility of the context component from the Strategy pattern to the CommandProcessor. There is no need for a separate LoggingContext object: the CommandProcessor can implement the invariant parts of the logging service easily. This integration of the Command Processor pattern with Strategy saves the first component of the original design.

The architecture can be cleaned-up further by combining Command with Composite. If the AbstractCommand class from the Command pattern also implements the role of the Composite pattern's component component, if ConcreteCommands are considered as leafs, and if a CompositeCommand component is added to this structure, the resulting design becomes less complex.

This new design is much easier to understand, maintain, and extend than the one we have originally developed. It is also more efficient, due to its less indirection levels.



However, in this design all four pattern implementations are tightly coupled and integrated. The design takes advantage of the relationships that exist between them. Command Processor and Composite complete Command, and Strategy completes Command Processor. Some components in the new design also play roles in several patterns. For example, the AbstractCommand class embodies the abstract command role of the Command and Command Processor patterns, and the component role of the Composite pattern. Likewise, the CommandProcessor is involved in the implementation of Command Processor, Command, and Strategy.

It All Comes Down to Roles!

Analyzing 'good' software architectures, those that are easy to understand, maintain, modify, extend, and adapt, and which are able to evolve with changing and growing requirements, will show that they expose a high density of tightly integrated patterns. Further analysis will reveal that it is actually this pattern density which enables these architectures' non-functional properties. Creating such architectures, however, is only possible if patterns expose manifold relationships among each other: that they complement and complete each other, and combine to larger structures in many different ways. In other words, the relationships between patterns are extremely important for their successful application in real-world software development.

Thus it is important to explicitly describe all relationships that a particular pattern has to other patterns: which patterns can take advantage of the pattern, which help with its implementation, and how do the referenced patterns integrate with each other.

Capturing these relationships as precisely as possible is one of the primary motivation behind the ongoing work on pattern and pattern languages. The goal is to understand patterns as connected nodes in a big network of relationships rather than individual islands waiting ages for their discovery. As we have demonstrated, a set of unrelated patterns is not half as worth as the same patterns with many relationships between them.

The property of patterns that allows us to take advantage of their manifold relationships is simple, and we have already stated it multiple times throughout this chapter: patterns introduce roles and responsibilities, rather than particular components or classes. These roles can be combined and integrated in many different ways without ever being twice the same, and without violating the essence of the patterns which participate in such an integrated structure.

Only when integrating roles developers can create designs which expose a high density of tightly integrated patterns. Only then they can build complex and heterogeneous software architectures with defined functional and non-functional properties using patterns [POSA1]. And only then patterns can help managing complexity in large software systems through providing well-working solutions to recurring analysis, design, and implementation problems. Otherwise software architectures built with patterns would become overly complex—both physically and logically [Lak95]—and would not expose their requested properties, as shown in this section. Patterns would be more a burden than a help, even if each resolves its own problem well. The notion of roles is thus where, regarding their implementation and practical use, the real power of patterns stems from.

1.6 A Reflection on the Reflection

The concept of patterns is hard. Even for experts in software development it took and still takes a long time to mine, understand, and communicate the idea. The original characterization, that a pattern is 'a solution to a problem in a context,' is not wrong. However, it is only a small piece of the whole cake. Patterns are much more than just this brief statement.

The intent of this chapter is to look much closer onto patterns than many common pattern discussions do, such as those in [POSA1], [GHJV95], [ADD COPE'S WHITE PAPER]. We did this by reflecting about the pattern concept and annotating existing pattern definitions, rather than by coining a new definition. Hopefully we can contribute with this reflection to the ongoing discussion on the characteristics of patterns and their practical use in real-world software development.

Nevertheless, it is likely that even more properties of patterns will be discovered and mined over time. This chapter only includes the insights *we* have gained until *today*. We therefore encourage you to reflect upon the concept of patterns on your own, and sharing your insights with the pattern community. This will help—step by step—to complete our all knowledge on patterns.