1 Patterns

...Somewhere in the deeply remote past it seriously traumatized a small random group of atoms drifting through the empty sterility of space and made them cling together in the most extraordinarily unlikely patterns. These patterns quickly learnt to copy themselves (this was part of what was so extraordinary about the patterns) and went on to cause massive trouble on every planet they drifted on to.
That was how life began in the Universe ...

Douglas Adams, The Hitchhiker's Guide to the Galaxy

Patterns help you build on the collective experience of skilled software engineers. They capture existing, well-proven experience in software development and help to promote good design practise. Every pattern deals with a specific, recurring problem in the design or implementation of a software system. Patterns can be used to construct software architectures with specific properties.

In this chapter we give an in-depth explanation of what patterns for software architecture are, and how they help you build software.

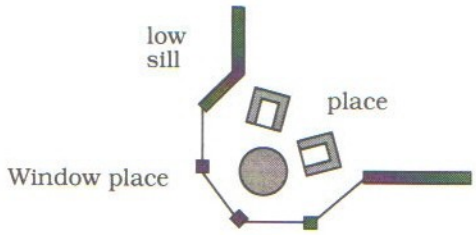
1.1 What is a Pattern?

When experts work on a particular problem, it is unusual for them to tackle it by inventing a new solution that is completely distinct from existing ones. They often recall a similar problem they have already solved, and reuse the essence of its solution to solve the new problem. This kind of 'expert behavior', the thinking in problem-solution pairs, is common to many different domains, such as architecture [Ale79], economics [Etz64] and software engineering [BJ94]. It is a natural way of coping with any kind of problem or social interaction [NS72].

Here is an elegant and intuitive example of such a problem-solution pair, taken from architecture:

Example **Window Place** [AIS77]:

> Everybody loves window seats, bay windows, and big windows with low sills and comfortable chairs drawn up to them...A room which does not have a place like this seldom allows you to feel comfortable or perfectly at ease...
>
> If the room contains no window which is a "place", a person in the room will be torn between two forces:
>
> 1. He wants to sit down and be comfortable.
>
> 2. He is drawn toward the light.
>
> Obviously, if the comfortable places—those places in the room where you most want to sit—are away from the windows, there is no way of overcoming this conflict...
>
> Therefore: In every room where you spend any length of time during the day, make at least one window into a "window place"



❑

Abstracting from specific problem-solution pairs and distilling out common factors leads to patterns: 'These problem-solution pairs tend to fall into families of similar problems and solutions with each family exhibiting a pattern in both the problems and the solutions' [Joh94]. In his book *The Timeless Way of Building* [Ale79] (p. 247), the architect Christopher Alexander defines the term *pattern* as follows:

> Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution.
>
> As an element in the world, each pattern is a relationship between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain spatial configuration which allows these forces to resolve themselves.
>
> As an element of language, a pattern is an instruction, which shows how this spatial configuration can be used, over and over again, to resolve the given system of forces, wherever the context makes it relevant.
>
> The pattern is, in short, at the same time a thing, which happens in the world, and the rule which tells us how to create that thing, and when we must create it. It is both a process and a thing; both a description of a thing which is alive, and a description of the process which will generate that thing.

We also find many patterns in software architecture. Experts in software engineering know these patterns from practical experience and follow them in developing applications with specific properties. They use them to solve design problems both effectively and elegantly. Before discussing this in detail, let us look at a well-known example:

Example Model-View-Controller (125)

Consider this pattern when developing software with a human-computer interface.

User interfaces are prone to change requests. For example, when extending the functionality of an application, menus have to be modified to access new functions, and user interfaces may have to be adapted for specific customers. A system may often have to be ported to another platform with a different 'look and feel' standard. Even upgrading to a new release of your window system can imply changes to your code. To summarize, the user interface of a long-lived system is a moving target.

Building a system with the required flexibility will be expensive and error-prone if the user interface is tightly interwoven with the functional core. This can result in the development and maintenance of several substantially different software systems, one for each user interface implementation. Ensuing changes then spread over many modules. In summary, when developing such an interactive software system, you have to consider two aspects:

- Changes to the user interface should be easy, and possible at run-time.
- Adapting or porting the user interface should not impact code in the functional core of the application.

To solve the problem, divide an interactive application into three areas: processing, output and input:

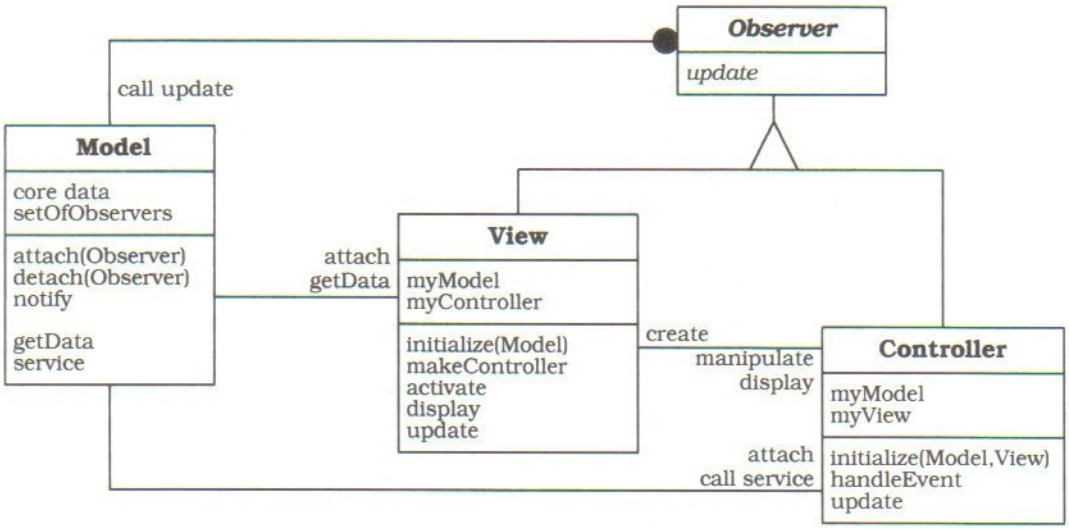
- The *model* component encapsulates core data and functionality. The model is independent of specific output representations or input behavior.
- *View* components display information to the user. A view obtains the data it displays from the model. There can be multiple views of the model.
- Each view has an associated *controller* component. Controllers receive input, usually as events that denote mouse movement, activation of mouse buttons or keyboard input. Events are translated to service requests, which are sent either to the model or to the view. The user interacts with the system solely via controllers.

The separation of the model from the view and controller components allows multiple views of the same model. If the user changes the model via the controller of one view, all other views dependent on this data should reflect the change. To achieve this, the model notifies all views whenever its data changes. The views in turn retrieve new data from the model and update their displayed information.

This solution allows you to change a subsystem of the application without causing major effects to other subsystems. For example, you can change from a non-graphical to a graphical user interface without modifying the model subsystem. You can also add support for a new input device without affecting information display or the functional

core. All versions of the software can operate on the same model subsystem independently of specific 'look and feel'.

The following OMT class diagram[1] illustrates this solution:



❑

We can derive several properties of patterns for software architecture from this introductory example[2]:

A pattern addresses a recurring design problem that arises in specific design situations, and presents a solution to it. In our example here the problem is supporting variability in user interfaces. This problem may arise when developing software systems with human-computer interaction. You can solve this problem by a strict separation of responsibilities: the core functionality of the application is separated from its user interface.

Patterns document existing, well-proven design experience. They are not invented or created artificially. Rather they 'distill and provide a means to reuse the design knowledge gained by experienced prac-

1. For a summary of the analysis and design method Object-Modeling-Technique (OMT) and its notation, see Notations on page 429. For details we refer to [RBPEL91].

2. If not stated otherwise, we use the terms *pattern* and *pattern for software architecture* as synonyms.

titioners' [GHJV93]. Those familiar with an adequate set of patterns 'can apply them immediately to design problems without having to rediscover them' [GHJV93]. Instead of knowledge existing only in the heads of a few experts, patterns make it more generally available. You can use such expert knowledge to design high-quality software for a specific task. The Model-View-Controller pattern, for example, presents experience gained over many years of developing interactive systems. Many well-known applications already apply the Model-View-Controller pattern—it is the classical architecture for many Smalltalk applications, and underlies several application frameworks such as MacApp [Sch86] or ET++ [WGM88].

Patterns identify and specify abstractions that are above the level of single classes and instances, or of components [GHJV93]. Typically, a pattern describes several components, classes or objects, and details their responsibilities and relationships, as well as their cooperation. All components together solve the problem that the pattern addresses, and usually more effectively than a single component. For example, the Model-View-Controller pattern describes a triad of three cooperating components, and each MVC triad also cooperates with other MVC triads of the system.

Patterns provide a common vocabulary and understanding for design principles [GHJV93]. Pattern names, if chosen carefully, become part of a widespread design language. They facilitate effective discussion of design problems and their solutions. They remove the need to explain a solution to a particular problem with a lengthy and complicated description. Instead you can use a pattern name, and explain which parts of a solution correspond to which components of the pattern, or to which relationships between them. For example, the name 'Model-View-Controller' and the associated pattern has been well-known to the Smalltalk community since the early '80s, and is used by many software engineers. When we say 'the architecture of the software follows Model-View-Controller', all our colleagues who are familiar with the pattern have an idea of the basic structure and properties of the application immediately.

Patterns are a means of documenting software architectures. They can describe the vision you have in mind when designing a software system. This helps others to avoid violating this vision when extending and modifying the original architecture, or when modifying

the system's code. For example, if you know that a system is structured according to the Model-View-Controller pattern, you also know how to extend it with a new function: keep core functionality separate from user input and information display.

Patterns support the construction of software with defined properties. Patterns provide a skeleton of functional behavior and therefore help to implement the functionality of your application. For example, patterns exist for maintaining consistency between cooperating components and for providing transparent peer-to-peer inter-process communication. In addition, patterns explicitly address non-functional requirements for software systems, such as changeability, reliability, testability or reusability. The Model-View-Controller pattern, for example, supports the changeability of user interfaces and the reusability of core functionality.

Patterns help you build complex and heterogeneous software architectures. Every pattern provides a predefined set of components, roles and relationships between them. It can be used to specify particular aspects of concrete software structures. Patterns 'act as building-blocks for constructing more complex designs' [GHJV93]. This method of using predefined design artifacts supports the speed and the quality of your design. Understanding and applying well-written patterns saves time when compared to searching for solutions on your own. This is not to say that individual patterns will necessarily be better than your own solutions, but, at the very least, a *pattern system* such as is explained in this book can help you to evaluate and assess design alternatives.

However, although a pattern determines the basic structure of the solution to a particular design problem, it does not specify a fully-detailed solution. A pattern provides a scheme for a generic solution to a family of problems, rather than a prefabricated module that can be used 'as is'. You must implement this scheme according to the specific needs of the design problem at hand. A pattern helps with the creation of similar units. These units can be alike in their broad structure, but are frequently quite different in their detailed appearance. Patterns help solve problems, but they do not provide complete solutions.

Patterns help you to manage software complexity. Every pattern describes a proven way to handle the problem it addresses: the kinds

of components needed, their roles, the details that should be hidden, the abstractions that should be visible, and how everything works. When you encounter a concrete design situation covered by a pattern there is no need to waste time inventing a new solution to your problem. If you implement the pattern correctly, you can rely on the solution it provides. The Model-View-Controller pattern, for example, helps you to separate the different user interface aspects of a software system and provide appropriate abstractions for them.

We end with the following definition:

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.

1.2 What Makes a Pattern?

The discussion in the previous section leads us to adopt a three-part schema that underlies every pattern:

Context: a situation giving rise to a problem.

Problem: the recurring problem arising in that context.

Solution: a proven resolution of the problem.

The schema as a whole denotes a type of rule that establishes a relationship between a given context, a certain problem arising in that context, and an appropriate solution to the problem. All three parts of this schema are closely coupled. However, to understand the schema in detail, we have to clarify what we mean by *context*, *problem*, and *solution*.

Context The context extends the plain problem-solution dichotomy by describing situations in which the problem occurs. The context of a

pattern may be fairly general, for example 'developing software with a human-computer interface.' On the other hand, the context can tie specific patterns together, such as 'implementing the change-propagation mechanism of the Model-View-Controller triad.'

Specifying the correct context for a pattern is difficult. We find it practically impossible to determine all situations, both general and specific, in which a pattern may be applied. A more pragmatic approach is to list all known situations where a problem that is addressed by a particular pattern can occur. This does not guarantee that we cover every situation in which a pattern may be relevant, but it at least gives valuable guidance.

Problem This part of a pattern description schema describes the problem that arises repeatedly in the given context. It begins with a general problem specification, capturing its very essence—what is the concrete design issue we must solve? The Model-View-Controller pattern, for example, addresses the problem that user interfaces often vary. This general problem statement is completed by a set of *forces*. Originally borrowed from architecture and Christopher Alexander, the pattern community uses the term *force* to denote any aspect of the problem that should be considered when solving it, such as:

- Requirements the solution must fulfil—for example, that peer-to-peer inter-process communication must be efficient.
- Constraints you must consider—for example, that inter-process communication must follow a particular protocol.
- Desirable properties the solution should have—for example, that changing software should be easy.

The Model-View-Controller pattern from the previous section specifies two forces: it should be easy to modify the user interface, but the functional core of the software should not be affected by its modification.

In general, forces discuss the problem from various viewpoints and help you to understand its details. Forces may complement or contradict each other. Two contradictory forces are, for example, extensibility of a system versus minimization of its code size. If you want your system to be extensible, you tend to use abstract superclasses. If you want to minimize code size, for example for

embedded applications, you may not be able to afford such a luxury as abstract superclasses. Most importantly, however, forces are the key to solving the problem. The better they are balanced, the better the solution to the problem. Detailed discussion of forces is therefore an essential part of the problem statement.

Solution The solution part of a pattern shows how to solve the recurring problem, or better, how to balance the forces associated with it. In software architecture such a solution includes two aspects.

Firstly, every pattern specifies a certain structure, a spatial configuration of elements. For example, the description of the Model-View-Controller pattern includes the following sentence: 'Divide an interactive application into the three areas: processing, output, and input.'

This structure addresses the *static* aspects of the solution. Since such a structure can be seen as a micro-architecture [GHJV93], it consists, like any software architecture, of both components and their relationships. Within this structure the components serve as building blocks, and each component has a defined responsibility. The relationships between the components determine their placement.

Secondly, every pattern specifies run-time behavior. For example, the solution part of the Model-View-Controller pattern includes the following statement: 'Controllers receive input, usually as events that denote mouse movement, activation of mouse buttons, or keyboard input. Events are translated to service requests, which are sent either to the model or to the view'.

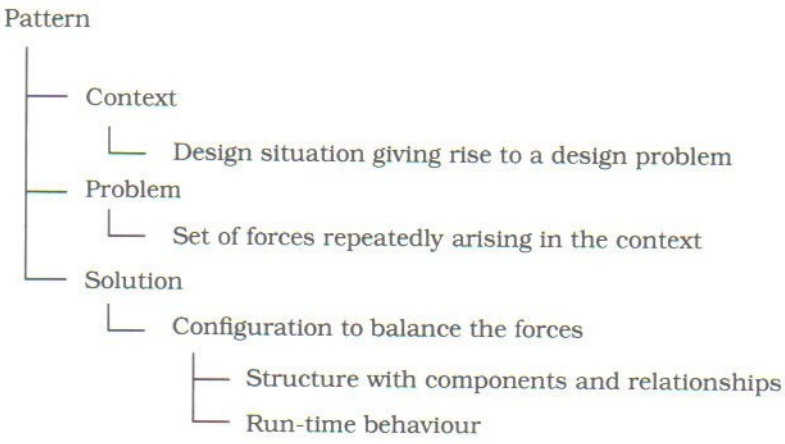
This run-time behavior addresses the *dynamic* aspects of the solution. How do the participants of the pattern collaborate? How is work organized between them? How do they communicate with each other?

It is important to note that the solution does not necessarily resolve all forces associated with the problem. It may focus on particular forces and leave others half or completely unresolved, especially if forces are contradictory.

As we mentioned in the previous section, a pattern provides a solution schema rather than a fully-specified artifact or blueprint. You should be able to reuse the solution in many implementations, but so that its essence is still retained. A pattern is a *mental* building block. After applying a pattern, an architecture should include a particular

structure that provides for the roles specified by the pattern, but adjusted and tailored to the specific needs of the problem at hand. No two implementations of a given pattern are likely to be the same.

The following diagram summarizes the whole schema:



This schema captures the very essence of a pattern independently of its domain. Using it as a template for describing patterns seems obvious. It already underlies many pattern descriptions, for example those in [AIS77], [BJ94], [Cope94c], [Cun94] and [Mes94]. This gives us confidence that the above form makes it easy to understand, share and discuss a pattern.

1.3 Pattern Categories

A closer look at existing patterns reveals that they cover various ranges of scale and abstraction. Some patterns help in structuring a software system into subsystems. Other patterns support the refinement of subsystems and components, or of the relationships between them. Further patterns help in implementing particular design aspects in a specific programming language. Patterns also range from domain-independent ones, such as those for decoupling interacting components, to patterns addressing domain-specific

aspects such as transaction policies in business applications, or call routing in telecommunication.

To refine our classification, we group patterns into three categories:

- Architectural patterns
- Design patterns
- Idioms

Each category consists of patterns having a similar range of scale or abstraction.

Architectural Patterns

Viable software architectures are built according to some overall structuring principle. We describe these principles with *architectural* patterns.

An *architectural pattern* expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them.

Architectural patterns are templates for concrete software architectures. They specify the system-wide structural properties of an application, and have an impact on the architecture of its subsystems. The selection of an architectural pattern is therefore a fundamental design decision when developing a software system.

The Model-View-Controller pattern from the beginning of this chapter is one of the best-known examples of an architectural pattern. It provides a structure for interactive software systems.

Design Patterns

The subsystems of a software architecture, as well as the relationships between them, usually consist of several smaller architectural units. We describe these using *design* patterns.

A *design pattern* provides a scheme for refining the subsystems or components of a software system, or the relationships between them. It describes a commonly-recurring structure of communicating components that solves a general design problem within a particular context [GHJV95].

Design patterns are medium-scale patterns. They are smaller in scale than architectural patterns, but tend to be independent of a particular programming language or programming paradigm. The application of a design pattern has no effect on the fundamental structure of a software system, but may have a strong influence on the architecture of a subsystem.

Many design patterns provide structures for decomposing more complex services or components. Others address the effective cooperation between them, such as the following pattern:

Name Observer [GHJV95] or Publisher-Subscriber (339)

Context A component uses data or information provided by another component.

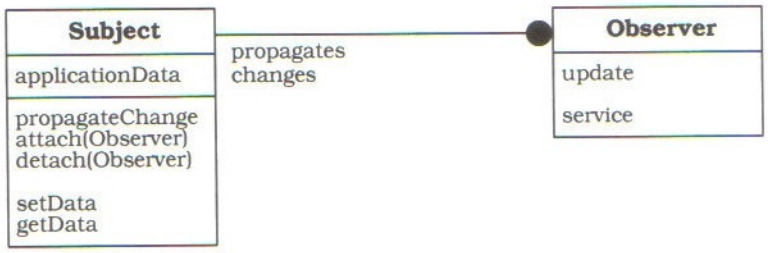
Problem Changing the internal state of a component may introduce inconsistencies between cooperating components. To restore consistency, we need a mechanism for exchanging data or state information between such components.

Two *forces* are associated with this problem:

- The components should be loosely coupled—the information provider should not depend on details of its collaborators.
- The components that depend on the information provider are not known a priori.

Solution Implement a change-propagation mechanism between the information provider—the *subject*—and the components dependent on it—the *observers*. Observers can dynamically register or unregister with this mechanism. Whenever the subject changes its state, it starts the change-propagation mechanism to restore consistency with all registered observers. Changes are propagated by invoking a special update

function common to all observers. To implement change propagation—the passing of data and state information from the subject to the observers—you can use a *pull-model*, a *push-model*, or a combination of both.



Idioms

Idioms deal with the implementation of particular design issues.

An *idiom* is a low-level pattern specific to a programming language. An idiom describes how to implement particular aspects of components or the relationships between them using the features of the given language.

Idioms represent the lowest-level patterns. They address aspects of both design and implementation.

Most idioms are language-specific—they capture existing programming experience. Often the same idiom looks different for different languages, and sometimes an idiom that is useful for one programming language does not make sense in another. For example, the C++ community uses reference-counting idioms to manage dynamically-allocated resources; Smalltalk provides a garbage collection mechanism, so has no need for such idioms.

The following example deals with a critical operation in C++: assignment. The pattern is called Counted Body. Its description is largely taken from [Cope94a]. We later describe the Counted Pointer pattern (353), which includes the Counted Body pattern as a variant.

Name Counted Body [Cope94a]

Context The interface of a class is separated from its implementation. A *handle* class presents the class interface to the user. The other class embodies the implementation, and is called *body*. The handle forwards member function invocations to the body.

Problem Assignment in C++ is defined recursively as member-by-member assignment, with copying as the termination of the recursion. In Smalltalk, it would be more efficient and more in the spirit of the language if copying were rebinding. In detail, you need to balance three *forces*:

- Copying of bodies is expensive in both storage requirements and processing time.
- Copying can be avoided by using pointers and references, but these leave a problem—who is responsible for cleaning up the object? They also leave a user-visible distinction between built-in types and user-defined types.
- Sharing bodies on assignment is semantically incorrect if the shared body is modified through one of the handles.

Solution A reference count is added to the body class to facilitate memory management. Memory management is added to the handle class, particularly to its implementations of initialization, assignment, copying and destruction. It is the responsibility of any operation that modifies the state of the body to break the sharing of the body by making its own copy, decrementing the reference count of the original body.

This solution avoids gratuitous copying, leading to a more efficient implementation. Sharing is broken when the body state is modified through any handle. Sharing is preserved in the more common case of parameter passing. Special pointer and reference types are avoided, and Smalltalk semantics are approximated. Garbage collection can be implemented based on this model.

Integration with Software Development

Ideally, our categories help you to preselect potentially useful patterns for a given design problem. They are related to important

software development activities. Architectural patterns can be used at the beginning of coarse-grained design, design patterns during the whole design phase, and idioms during the implementation phase. A more detailed discussion of these issues can be found in Section 5.2, *Pattern Classification*, together with a discussion of alternative classification schemas.

1.4 Relationships between Patterns

A close look at many patterns reveals that, despite initial impressions, their components and relationships are not always as 'atomic' as they first appear to be. A pattern solves a particular problem, but its application may raise new problems. Some of these can be solved by other patterns. Single components or relationships inside a particular pattern may therefore be described by smaller patterns, all of them integrated by the larger pattern in which they are contained.

Example Refinement of the Model-View-Controller pattern

The Model-View-Controller pattern separates core functionality from human-computer interaction to provide adaptable user interfaces. However, applying this pattern introduces a new problem. Views, and sometimes even controllers, depend on the state of the model. The consistency between them must be maintained: whenever the state of the model changes, we must update all its dependent views and controllers. However, we must not lose the ability to change the user interface. The Observer pattern from the previous section helps us to solve this problem—the model embodies the role of the subject, while views and controllers play the roles of observers. ❑

Most patterns for software architecture raise problems that can be solved by smaller patterns. Patterns do not usually exist in isolation. Christopher Alexander puts this in somewhat idealistic terms: 'Each pattern depends on the smaller patterns it contains and on the larger patterns in which it is contained' [Ale79].

A pattern may also be a variant of another. From a general perspective a pattern and its variants describe solutions to very similar problems.

These problems usually vary in some of the forces involved, rather than in general character. This is illustrated in the following example.

Example The Document-View variant of the Model-View-Controller pattern.

Consider the development of an interactive text editor using the Model-View-Controller pattern. Within such an application it is hard to separate controller functionality from view functionality. Suppose you select text with the mouse and change it from regular to bold face. Text selection is a controller action that does not cause changes to the model. The selected text just serves as input for another controller action, here changing the face of the selected text. However, text selection has a visual appearance—the selected text is highlighted. In a strict Model-View-Controller structure, the controller must either implement this 'view-like' behavior by itself, or must cooperate with the view in which the selected text appears. Both solutions require some unnecessary implementation overhead.

In such a situation it is better to apply the Document-View variant of the Model-View-Controller pattern, which unifies the view and controller functionality in a single component, the view of the Document-View pattern. The document component directly corresponds to the model of the Model-View-Controller triad. When using the Document-View variant, however, we lose the ability to change input and output functionality independently. ❑

Patterns can also combine in more complex structures at the same level of abstraction. This happens when your original problem includes more forces than can be balanced by a single pattern. In this case, applying several patterns can solve the problem. Each pattern resolves a particular subset of the forces.

Example Transparent peer-to-peer inter-process communication

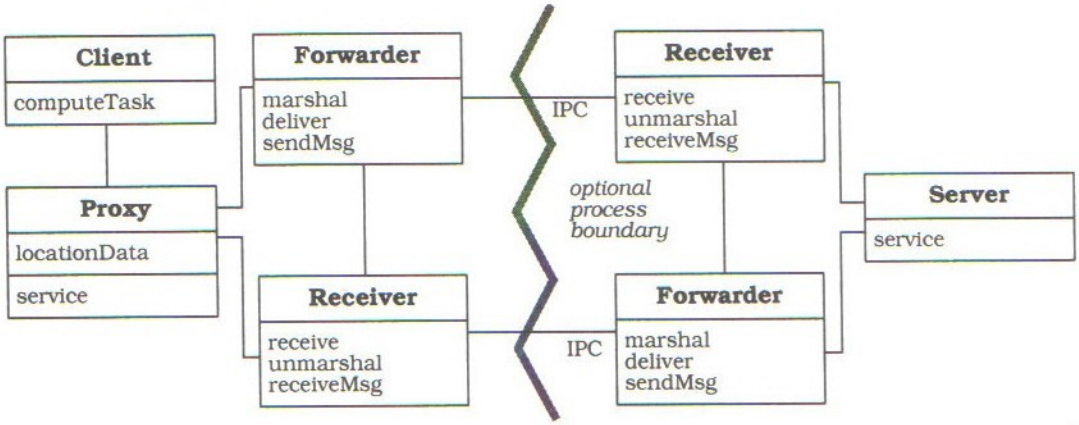
Suppose you have to develop a distributed application with high performance peer-to-peer inter-process communication. The following *forces* must be balanced:

- The inter-process communication must be efficient. Spending time searching for the location of remote servers is undesirable.
- Independence from a particular inter-process communication mechanism is desirable. The mechanism must be exchangeable without affecting clients or servers.

- Clients should not be aware of, or dependent on, the name and location of their servers. Instead, they should communicate with each other as if they were in the same process.

This problem cannot be solved by any single pattern in isolation, but two patterns in combination can achieve this. The Forwarder-Receiver pattern (307) resolves the first and second force. It offers a general interface for sending and receiving messages and data across process boundaries. The pattern hides the details of the concrete inter-process communication mechanism. Replacing this mechanism only affects the forwarders and receivers of the system. In addition, the pattern offers a name-to-address mapping for servers.

The Proxy pattern (263) resolves the third force. In this pattern, the client communicates with a representative of the server that is located in the same process. This representative, the *remote proxy*, knows details about the server, such as its name, and forwards every request to it.



❑

All three kinds of relationship—refinement, variants and combination—help in using patterns effectively. Refinement supports the implementation of a pattern, combination helps you compose complex design structures, and variants help when selecting the right pattern in a given design situation.

You can find complementary discussion of relationships between patterns in [Zim94].

1.5 Pattern Description

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

Patterns should also be described uniformly. This helps us to compare one pattern with another, especially when we are looking for alternative solutions to a problem.

The basic Context-Problem-Solution structure we discussed earlier in this chapter provides us with a good starting point for a description format that meets the above requirements. It captures the essential characteristics of a pattern, and provides you with the key ideas. We have therefore based our description template on this structure.

However, describing a pattern based exclusively on a Context-Problem-Solution schema is not enough. A pattern must be named—preferably with an intuitive name—if we are to share it and discuss it. Such a name should also convey the essence of a pattern. A good pattern name is vital, as it will become part of the design vocabulary [GHJV93].

We add an introductory example to the pattern description to help explain the problem and its associated forces. We repeatedly refer to this example when discussing solution and implementation aspects of the general pattern.

We further use diagrams and scenarios to illustrate the static and dynamic aspects of the solution. We also include implementation guidelines for the pattern. These guidelines help us transform a given architecture into one that uses the pattern. We add sample code, and list successful applications of the pattern to enhance its credibility.

We also describe variants of a pattern. Variants provide us with alternative solutions to a problem. However, we do not describe these variants at the same level of detail as the original pattern—we only describe them briefly.

A discussion of the benefits and potential liabilities of a pattern highlight the consequences of its application. This provides us with information to help us decide whether we can use the pattern to provide an adequate solution to a specific problem. We also cross-reference other related patterns, either because they refine the current pattern, or because they address a similar problem.

With all this information available and appropriately laid out, we should be able to understand a pattern, apply and implement it correctly.

Finally we give credits to all who helped to shape a particular pattern. Writing patterns is hard. Achieving a crisp pattern description takes several review and revision cycles. Many experts from all over the world have helped us with this activity, and we owe them our special thanks. If we know the names of the pattern 'discoverers', the persons who originally described the pattern, we give credits to them as well.

Our pattern description template is therefore as follows:

Name The name and a short summary of the pattern.

Also Known As Other names for the pattern, if any are known.

Example A real-world example demonstrating the existence of the problem and the need for the pattern.

Throughout the description we refer to the example to illustrate solution and implementation aspects, where this is necessary or useful. Text that is specifically about the example is marked by the ➥ symbol at its beginning and by the ❑ symbol at its end.

Context The situations in which the pattern may apply.

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

Solution The fundamental solution principle underlying the pattern.

Structure A detailed specification of the structural aspects of the pattern, including CRC-cards [BeCu89] (see Notations on page 429) for each participating component and an OMT class diagram [RBPEL91].

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

We further illustrate the scenarios with Object Message Sequence Charts (see Notations on page 431).

Implementation Guidelines for implementing the pattern.

These are only a suggestion, not an immutable rule. You should adapt the implementation to meet your needs, by adding different, extra, or more detailed steps, or by re-ordering the steps. We give C++, Smalltalk, Java or pSather code fragments to illustrate a possible implementation, often describing details of the example problem.

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

Variants A brief description of variants or specializations of a pattern.

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

Consequences The benefits the pattern provides, and any potential liabilities.

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

1.6 Patterns and Software Architecture

An important criterion for the success of patterns is how well they meet the objectives of software engineering. Patterns must support the development, maintenance and evolution of complex, large-scale systems. They must also support effective industrial software production, otherwise they remain just an interesting intellectual concept, but useless for constructing software.

Patterns as Mental Building-Blocks

We have already learned that patterns are useful mental building-blocks for dealing with limited and specific design aspects when developing a software system.

Patterns therefore address an important objective of software architecture—the construction of specific software architectures with defined properties. Consider the Model-View-Controller pattern again. It provides a structure that supports the tailoring of the user interface of an interactive application.

General techniques for software architecture, such as guidelines on using object-oriented features such as inheritance and polymorphism, do not address the solution of specific problems. Most of the existing analysis and design methods also fail at this level. They only provide general techniques for building software, for example 'separate policy from implementation' [RBPEL91]. The creation of specific architectures is still based on intuition and experience.

Patterns effectively complement these general problem-independent architectural techniques with specific problem-oriented ones. Note that patterns do not make existing approaches to software architecture obsolete—instead, they fill a gap that is not covered by existing techniques.

Constructing Heterogenous Architectures

A single pattern cannot enable the detailed construction of a complete software architecture—it just helps you to design one aspect of your application. Even if you design one aspect correctly, however, the whole architecture may still fail to meet its desired overall properties. To meet the needs of software architecture 'in the large' we need a rich set of patterns that must cover many different design problems. The more patterns that are available, the more design problems that can be addressed appropriately, and the more we are supported in constructing software architectures with defined properties.

On the other hand, the more patterns that are available, the harder it is to achieve an overview of them. As we have already pointed out, there are many relationships between patterns. When applying one pattern, you want to know which other patterns can help refine the structure it introduces. You also want to know which other patterns you can combine with it.

To use patterns effectively, we therefore need to organize them into *pattern systems*. A pattern system describes patterns uniformly, classifies them, and most importantly, shows how they are interwoven

with each other. Pattern systems also help you to find the right pattern to solve a problem or to identify alternative solutions to it. This is in contrast to a *pattern catalog*, where each pattern is described more or less in isolation from other patterns. Pattern systems help us to use the power that the entirety of patterns provides.

Patterns versus Methods

A good pattern description also includes guidelines for its implementation that you can consider as a *micro-method* for creating the solution to a specific problem. These micro-methods complement general but problem-independent analysis and design methods, such as Booch [Boo94] and Object Modeling Technique [RBPEL91], by providing methodological steps for solving concrete recurring problems in software development. Section 5.4, *Pattern Systems as Implementation Guidelines* discusses this issue in detail.

Implementing Patterns

Another aspect that arises from the integration of patterns with software architecture is a paradigm for implementing them. Many current software patterns have a distinctly object-oriented flavor. It is tempting to conclude that the only way we can implement a pattern effectively is in an object-oriented programming language. However, we think such conclusions are false.

On one hand, it is true that many patterns, including those in this book, use object-oriented techniques such as polymorphism and inheritance. Examples of such patterns are the Strategy pattern [GHJV95] and the Proxy pattern (263).

On the other hand, object-oriented features are not essential for implementing these patterns. Proxy, for example, loses only a small fraction of its elegance by giving up inheritance. Strategy can be implemented in C by using function pointers instead of polymorphism and inheritance.

At the design level, most patterns only require certain abstraction facilities of a programming language, such as modules or data abstraction. You can therefore implement patterns with almost any programming paradigm and in almost any programming language. In

addition, every programming language has specific patterns of its own, the idioms of that language. They capture existing programming experience with the language and define a programming style for it.

In conclusion, we can say that there is no single paradigm or language for implementing patterns. Patterns can be integrated with every paradigm used for constructing software architectures.

1.7 Summary

Patterns provide a promising approach for developing software with defined properties. They document existing design knowledge and help you find appropriate solutions to design problems. Patterns exist in various ranges of scale and abstraction, and cover many different and important areas of software development. Patterns are interwoven with each other—you can use them to refine other, larger patterns and you can combine them to solve more complex problems. They address important aspects of software architecture and complement existing techniques and methods. You can integrate them with every programming paradigm and implement them in almost any programming language. In summary, the entirety of patterns provides a *mental toolbox* that helps you construct software that meets both the functional and non-functional requirements of an application.

Patterns are already being successfully applied. We find them in applications from the business domain [EKM+94], the automation domain [BM95] and the telecommunication domain [Sch95]. They play an important role in application frameworks such as ET++ [WGM88] or InterViews [LCITV92], as well as in run-time environments like the Meta-Information-Protocol for C++ [BKSP92].

To exploit the full power of patterns, however, we need to provide technical and methodical support that goes beyond the scope of individual patterns. We address some of these aspects in Chapter 5, *Pattern Systems*.