Chapter 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 Sys-*

tems, 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 nonfunctional 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.

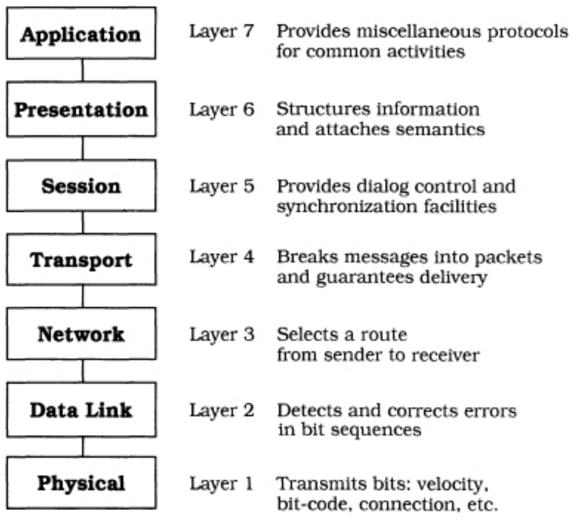
The Blackboard pattern comes from the Artificial Intelligence community. We describe this paradigm as a pattern since the idea behind it deserves to be seen in a wider context. In poorly-structured—or simply new and immature—domains we often have only patchy knowledge about how to tackle particular problems. 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 sub-protocols 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]:


Application
Layer 7
Provides miscellaneous protocols for common activities
Presentation
Layer 6
Structures information and attaches semantics
Session
Layer 5
Provides dialog control and synchronization facilities
Transport
Layer 4
Breaks messages into packets and guarantees delivery
Network
Layer 3
Selects a route from sender to receiver
Data Link
Layer 2
Detects and corrects errors in bit sequences
Physical
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

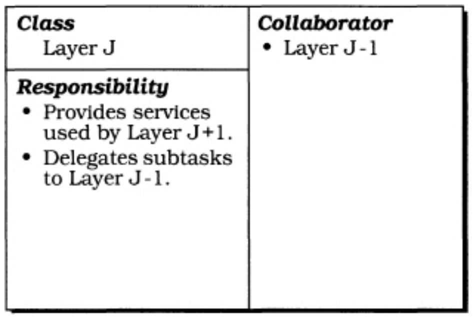
From a high-level viewpoint the solution is extremely simple. Structure your system into an appropriate number of layers and place them on top of each other. Start at the lowest level of abstraction—call it Layer 1. This is the base of your system. Work your way up the abstraction ladder by putting Layer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

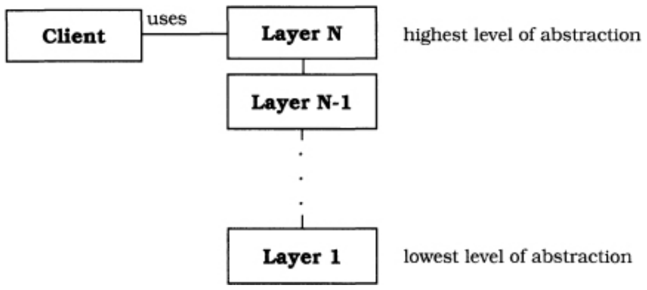
Structure

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

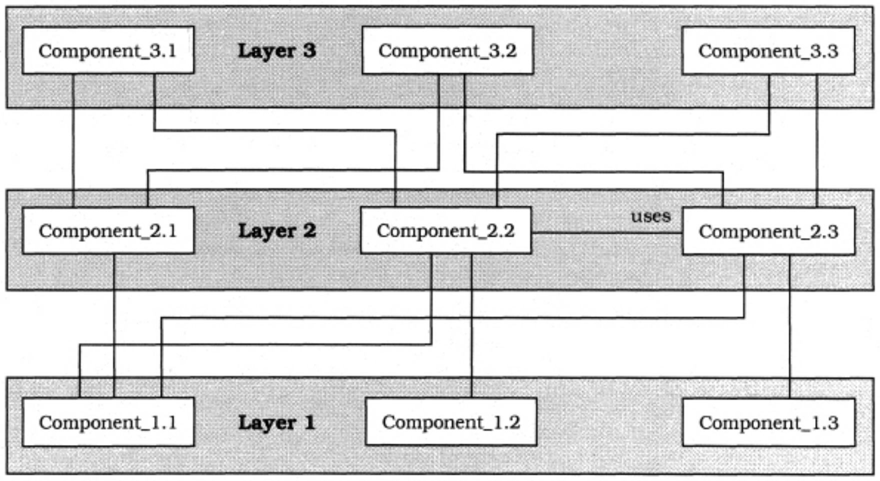


Class Layer J	***Collaborator***
Responsibility • Provides services used by Layer J+1. • Delegates subtasks to Layer J-1.	• Layer J-1

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.

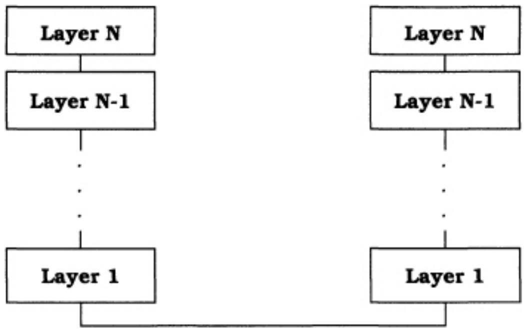
Scenario II illustrates bottom-up communication—a chain of actions starts at Layer 1, for example when a device driver detects input. The driver translates the input into an internal format and reports it to Layer 2, which starts interpreting it, and so on. In this way data moves up through the layers 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+l 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 callback 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 *12) {leve12 = 12;}
protected:
    L2Provider *leve12;
};

class DataLink : public L1Provider {
public:
    virtual void L1Service() {
        cout << “L1Service doing its job” << end1;}
};
class Transport : public L2Provider {
public:
    virtual void L2Service() {
        cout << “L2Service starting its job” << end1;
        level1->L1Service ();
        cout << “L2Service finishing its job” << end1;}
};
class Session : public L3Provider {
public:
    virtual void L3Service() {
        cout << “L3Service starting its job” << end1;
        level2->L2Service();
        cout << “L3Service finishing its job” << end1;}
};

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

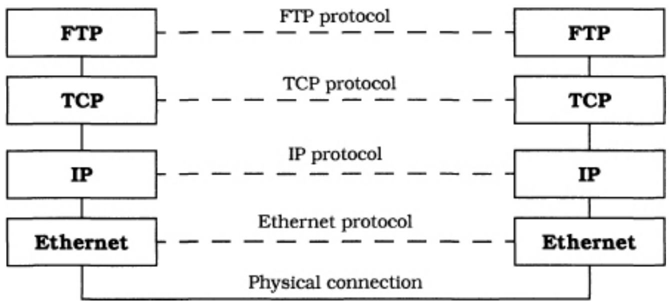
`L3Provider` and `L2Parent`.

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 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.
- 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 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

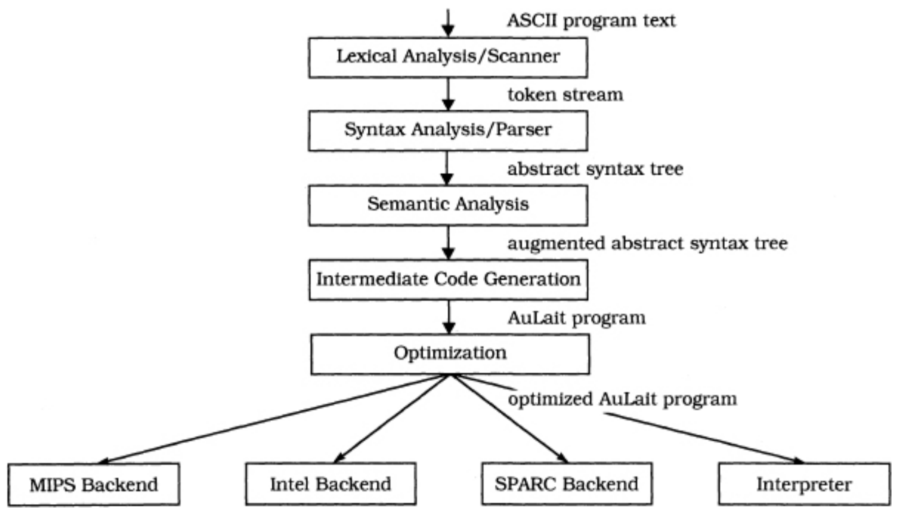
This pattern was carefully reviewed by Paulo Villela, who highlighted many dark corners in earlier drafts. Douglas Schmidt gave valuable support in the ATM discussion.

Pipes and Filters

The *Pipes and Filters* architectural pattern 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.

Example

Suppose we have defined a new programming language called Mocha (Modular Object Computation with Hypothetical Algorithms). Our task is to build a portable compiler for this language. To support existing and future hardware platforms we define an intermediate language AuLait (Another Universal Language for Intermediate Translation) running on a virtual machine Cup (Concurrent Uniform Processor). Cup will be implemented by an interpreter or platform-specific backends. The AuLait interpreter simulates Cup in software. A backend will translate AuLait code into the machine instructions of a specific processor for best performance.



Conceptually, translation from Mocha to AuLait consists of the phases lexical analysis, syntax analysis, semantic analysis, intermediate-code generation (AuLait), and optionally intermediate-code optimization [ASU86]. Each stage has well-defined input and output data. The input to the compilation process is a sequence of ASCII characters representing the Mocha program. The final stage in our system—whether backend or interpreter—takes the binary AuLait code as its input.[4]

Context

Processing data streams.

Problem

Imagine you are building a system that must process or transform a stream of input data. Implementing such a system as a single component may not be feasible for several reasons: the system has to be built by several developers, the global system task decomposes naturally into several processing stages, and the requirements are likely to change.

You therefore plan for future flexibility by exchanging or reordering the processing steps. By incorporating such flexibility, it is possible to build a family of systems using existing processing components. The design of the system—especially the interconnection of processing steps—has to consider the following *forces:*

- Future system enhancements should be possible by exchanging processing steps or by recombination of steps, even by users.
- Small processing steps are easier to reuse in different contexts than large components.
- Non-adjacent processing steps do not share information.
- Different sources of input data exist, such as a network connection or a hardware sensor providing temperature readings, for example.
- It should be possible to present or store final results in various ways.
- Explicit storage of intermediate results for further processing in files clutters directories and is error-prone, if done by users.
- You may not want to rule out multi-processing the steps, for example running them in parallel or quasi-parallel.

Whether a separation into processing steps is feasible strongly depends on the application domain and the problem to be solved. For example, an interactive, event-driven system does not split into sequential stages.

Solution

The Pipes and Filters architectural pattern divides the task of a system into several sequential processing steps. These steps are connected by the data flow through the system—the output data of a step is the input to the subsequent step. Each processing step is implemented by a *filter* component. A filter consumes and delivers data incrementally—in contrast to consuming all its input before producing any output—to achieve low latency and enable real parallel processing. The input to the system is provided by a *data source* such as a text file. The output flows into a *data sink* such as a file, terminal, animation program and so on. The data source, the filters and the data sink are connected sequentially by *pipes.* Each pipe implements the data flow between adjacent processing steps. The sequence of filters combined by pipes is called a *processing pipeline.*

Structure

Filter components are the processing units of the pipeline. A filter enriches, refines or transforms its input data. It enriches data by computing and adding information, refines data by concentrating or extracting information, and transforms data by delivering the data in some other representation. A concrete filter implementation may combine any of these three basic principles.

The activity of a filter can be triggered by several events:

- The subsequent pipeline element pulls output data from the filter.
- The previous pipeline element pushes new input data to the filter.
- Most commonly, the filter is active in a loop, pulling its input from and pushing its output down the pipeline.

The first two cases denote so-called passive filters, whereas the last case is an active filter[5]. An active filter starts processing on its own as a separate program or thread. A passive filter component is activated by being called either as a function (pull) or as a procedure (push).

Pipes denote the connections between filters, between the data source and the first filter, and between the last filter and the data sink. If two active components are joined, the pipe synchronizes them. This synchronization is done with a first-in-first-out buffer. If activity is controlled by one of the adjacent filters, the pipe can be implemented by a direct call from the active to the passive component. Direct calls make filter recombination harder, however.

Class Filter	***Collaborators*** • Pipe
Responsibility • Gets input data. • Performs a function on its input data. • Supplies output data.	

Class Pipe	***Collaborators*** • Data Source • Data Sink • Filter
Responsibility • Transfers data. • Buffers data. • Synchronizes active neighbors.	

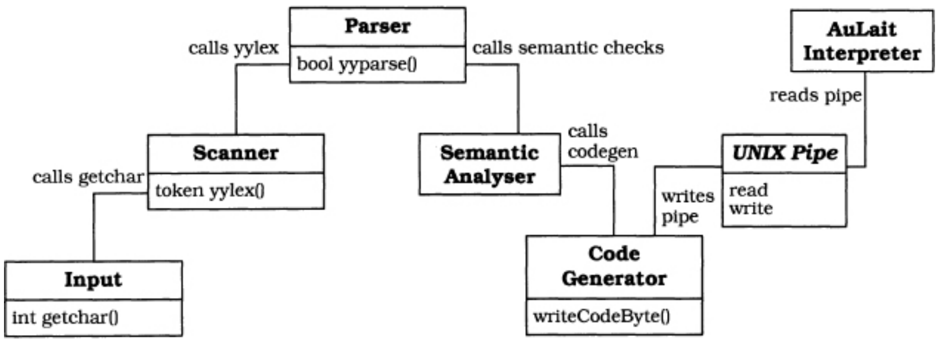
The *data source* represents the input to the system, and provides a sequence of data values of the same structure or type. Examples of such data sources are a file consisting of lines of text, or a sensor delivering a sequence of numbers. The data source of a pipeline can either actively push the data values to the first processing stage, or passively provide data when the first filter pulls.

The *data sink* collects the results from the end of the pipeline. Two variants of the data sink are possible. An active data sink pulls results out of the preceding processing stage, while a passive one allows the preceding filter to push or write the results into it.

Class Data Source	***Collaborators*** • Pipe
Responsibility • Delivers input to processing pipeline.	

Class Data Sink	***Collaborators*** • Pipe
Responsibility • Consumes output.	

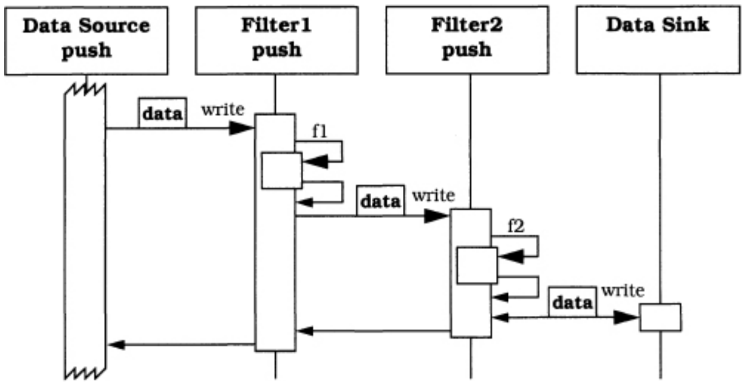
➥ In our Mocha compiler we use the UNIX tools `lex` and `yacc` to implement the first two stages of the compiler [ASU86]. Both tools generate functions—`yylex()` and `yyparse()`—for embedding in a program. The function `yyparse()` actively controls the frontend of our compiler. It calls `yylex()` whenever further input tokens are needed. The connection to the other frontend stages consists of many procedure calls embedded in the grammar action rules, and not just simple data flow. Such embedded calls are more efficient than creating an explicit abstract syntax tree representation and passing it along a pipe. The backends and interpreter run as separate programs to allow exchangeability. They are connected via a UNIX pipe to the frontend.



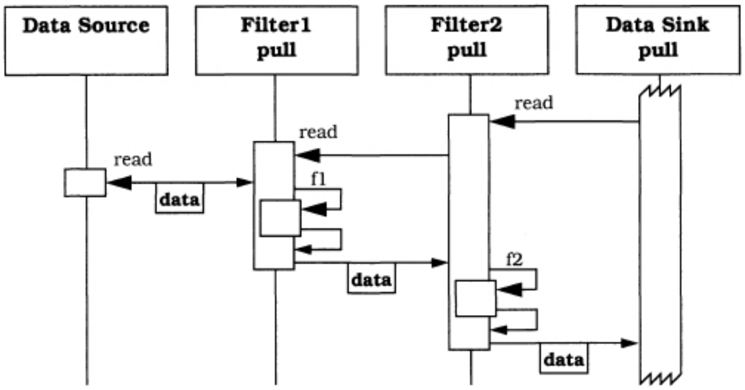
Dynamics

The following scenarios show different options for control flow between adjacent filters. Assume that `Filter1` computes function `f1` on its input data and `Filter2` function `f2`. The first three scenarios show passive filters that use direct calls to the adjacent pipeline components, with different components controlling the activity—no explicit pipe components therefore exist. The last scenario shows the commonest case, in which all filters are active, with a synchronizing pipe between them.

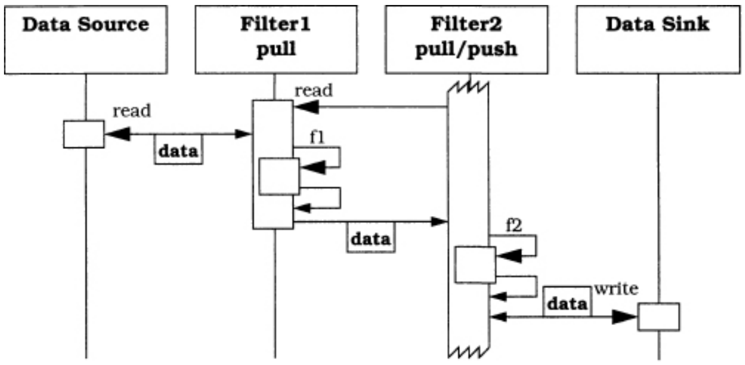
Scenario I shows a push pipeline in which activity starts with the data source. Filter activity is triggered by writing data to the passive filters.



Scenario II shows a pull pipeline. Here control flow is started by the data sink calling for data.



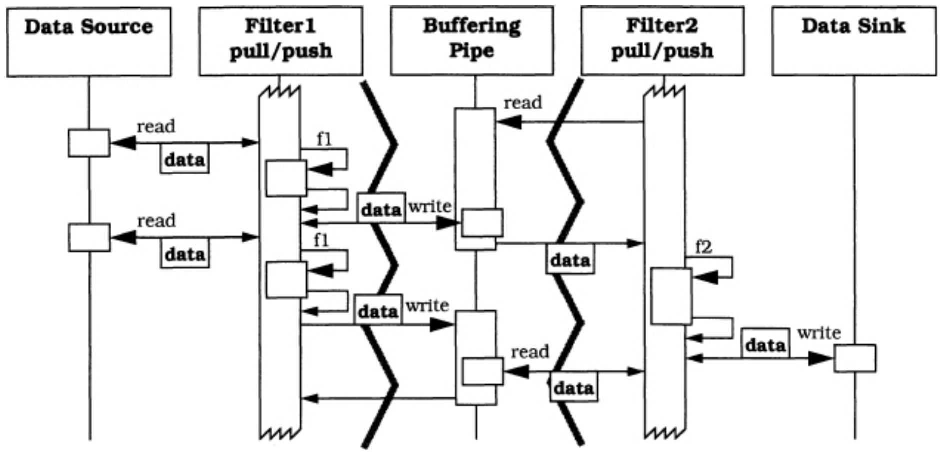
Scenario III shows a mixed push-pull pipeline with passive data source and sink. Here the second filter plays the active role and starts the processing.



Scenario IV shows a more complex but typical behavior of a Pipes and Filters system. All filters actively pull, compute, and push data in a loop. Each filter therefore runs in its own thread of control, for example as a separate process. The filters are synchronized by a buffering pipe between them. For simplicity we assume that the pipe buffers only a single value. This scenario also shows how you can achieve parallel execution using filters.

The following steps occur in this scenario:

- `Filter2` tries to get new data by reading from the pipe. Because no data is available the data request suspends the activity of `Filter2`—the buffer is empty.
- `Filter1` pulls data from the data source and performs function `f1`.
- `Filter1` then pushes the result to the pipe.
- `Filter2` can now continue, because new input data is available. `Filter1` can also continue, because it is not blocked by a full buffer within the pipe.
- `Filter2` computes `f2` and writes its result to the data sink.
- In parallel with `Filter2`'s activity, `Filterl` computes the next result and tries to push it down the pipe. This call is blocked because `Filter2` is not waiting for data—the buffer is full.
- `Filter2` now reads new input data that is already available from the pipe. This releases `Filter1` so that it can now continue its processing.



Implementation

Implementing a Pipes and Filters architecture is straightforward. You can use a system service such as message queues or UNIX pipes for pipe connections, or other options like the direct call implementation, as described in steps 3 through 6 below. The design decisions in these steps are closely interrelated, so you may make them in an order other than that given here. The implementation of the data source and data sink is not addressed explicitly, because it follows the guidelines for pipes or filters closely.

1 *Divide the system's task into a sequence of processing stages.* Each stage must depend only on the output of its direct predecessor. All stages are conceptually connected by the data flow. If you plan to develop a family of systems by exchanging processing stages, or if you are developing a toolbox of components, you can consider alternatives or recombinations for some processing stages at this point.
➥ In our Mocha compiler the primary separation is between the AuLait-creating frontend and the backends. Further structuring of the frontend gives us the stages of scanner, parser, semantic analyzer and code generator. We decide not to construct an abstract syntax tree explicitly, to be passed from the parser to the semantic analyzer. Instead we embed calls to the semantic analyzer `(sa)` and code generator

(cg) into yacc's grammar rules:

```
addexpr : term
     | addexpr '+' term
         { sa.checkCompat($1,$3); cg.genAdd($1,$3); }
     | addexpr '-' term
         { sa.checkCompat($1,$3); cg.genSub($1,$3); }
```

This means that we need to build a filter consisting of the parser, semantic analyzer and code generator stages. The scanner, as a separate filter component, remains passive until called by the parser. We thus link the function `yylex()` into our frontend program.

2 *Define the data format to be passed along each pipe.* Defining a uniform format results in the highest flexibility because it makes recombination of filters easy. In most UNIX filter programs the data format is line-structured ASCII text. This may however impose an efficiency penalty. For example, a textual representation of floating-point numbers may be too inefficient to pass along a pipe, because repeated conversion between ASCII and floating-point representations and back is needed. If you both want flexibility and opt for different data representations, you can create transformation filter components to change the data between semantically-equivalent representations.

You must also define how the end of input is marked. If a system service is used for pipe connections an end-of-input error condition may be sufficient. For other pipe implementations you can use a special data value to mark the end of input. The values zero, -1, $, control-D, or control-Z are favored examples of end-of-input markers.

➥ The input to our frontend is a Mocha program, in the form of a stream or file of ASCII characters. The tokens passed from scanner to parser are denoted by integer values. The function `yylex()` returns either the ASCII code of a character scanned, or a code beyond the ASCII range for tokens, such as a Mocha keyword. The end of input is marked by the value zero. The data format used between the frontend and the backends or interpreter is provided by the definition of the AuLait byte codes.

3 *Decide how to implement each pipe connection.* This decision directly determines whether you implement the filters as active or passive components. Adjacent pipes further define whether a passive filter is triggered by push or pull of data. The simplest case of a pipe connection is a direct call between adjacent filters to push or pull a data value, as shown in the first three scenarios of the Dynamics section. If you use a direct call between filters, however, you have to change your code whenever you want to recombine or reuse filter components. Such filters are also harder to develop and test in isolation, due to the need for test frames to call the filter components.

Using a separate pipe mechanism that synchronizes adjacent active filters provides a more flexible solution. If all pipes use the same mechanism, arbitrary recombination of filters becomes possible. A pipe supplies a first-in-first-out buffer to connect adjacent filters that produce and consume unequal amounts of data per computation. Many operating systems provide inter-process communication services such as queues or pipes that you can use to connect active filter programs. If such services are not available, you can implement filters as separate threads, and pipes as queues that synchronize producers and consumers of data.

➥ Because we want flexibility at the backend of our Mocha compiler, we use the UNIX pipe mechanism between frontend and backends. This also allows us to store the intermediate results of our compilation —the AuLait code—in a file for further analysis or translation by another backend.

4 *Design and implement the filters.* The design of a filter component is based both on the task it must perform and on the adjacent pipes. You can implement passive filters as a function, for pull activation, or as a procedure for push activation. Active filters can be implemented either as processes or as threads in the pipeline program.

The cost of a context switch between processes, and the need to copy data between address spaces, may heavily impact performance. The buffer size of the pipes is an additional parameter you should take into account. A small buffer gives the worst case when combined with the most context switches. You can achieve high flexibility with small active filter components at the price of an overhead for many context switches and data transfers.

If you want to be able to reuse filters easily, it is vital to control their behavior in some way. Several techniques are available for passing parameters to filters. UNIX filter programs, for example, allow many options to be passed on the command line. An alternative method is to use a global environment or repository that is available to filters when they execute. This can be supported by the operating system, the shell or a configuration file. You should think carefully about the trade-off between the flexibility of a filter and its ease of use. As a rule of thumb, a filter should do one thing well.
➥ Our Mocha frontend reads program source code from standard input and creates an AuLait program on its standard output. The stages within the frontend communicate by direct calls. We also create an optimizer for AuLait running as a separate filter program. The AuLait interpreter can be viewed as a data sink, whereas the intended backends are additional filter stages producing object code as output.
5 *Design the error handling.* Because pipeline components do not share any global state, error handling is hard to address and is often neglected. As a minimum, error detection should be possible. UNIX defines a specific output channel for error messages, `stderr`, that is used by most provided filter programs for this purpose. Such an approach can denote errors in input data, resource limitations and so on. A single error channel may however mix error messages from different components in a non-obvious and unpredictable way when filters run in parallel.
If a filter detects errors in its input data, it can ignore input until some clearly marked separation occurs. For example, a filter may skip to the next line of input if a line is expected to contain a numerical value and does not. This approach is helpful if incorrect input data is possible and inaccurate results can be tolerated.
It is hard to give a general strategy for error handling with a system based on the Pipes and Filters pattern. For example, consider the case in which a pipeline has consumed three-quarters of its input, already produced half of its output data, and some intermediate filter crashes. In many systems the only solution is to restart the pipeline and hope that it will complete without failure.
Resynchronization of the pipeline can be a goal of an advanced system in which filters process data incrementally. One option is to introduce special marker values to tag the input data stream. These markers are passed unchanged to the output. You cam then restart the pipe at the correct stage of the input to continue processing from a failure. Another option is to use pipes to buffer data that has already been consumed, and then use it to restart the pipeline if a filter crashes.
➥ In our simple compiler we send errors to the standard error channel. The parser is designed to skip tokens when it detects a syntax error until the scanner recognizes the ‘;’ statement separator. This is done for example in the following grammar rule, which ignores syntax errors if they occur in an ‘import’ statement.

```
import : FROM identifier objidentlist ‘;’
       | FROM error ‘;’
            { mochaerror(errs[E_IMPRT]); yyerrok; }
```

In this rule `yacc`’s special token `error` matches all unrecognized tokens until a semicolon is found. The statement `yyerrok` is a special action that resets the parser into normal mode after the occurrence of a syntax error.
6 *Set up the processing pipeline.* If your system handles a single task you can use a standardized main program that sets up the pipeline and starts processing. This type of system may benefit from a direct-call pipeline, in which the main program calls the active filter to start processing.
You can increase flexibility by providing a shell or other end-user facility to set up various pipelines from your set of filter components. Such a shell can support the incremental development of pipelines by allowing intermediate results to be stored in files, and supporting files as pipeline input. You are not restricted to a text-only shell such as those provided by UNIX, and could even develop a graphical environment for visual creation of pipelines using ‘drag and drop’ interaction.
➥ Our compiler is set up by a UNIX shell command that establishes the compilation or interpreter pipeline:

```
    # compile and optimize a Mocha program for a Sun
```

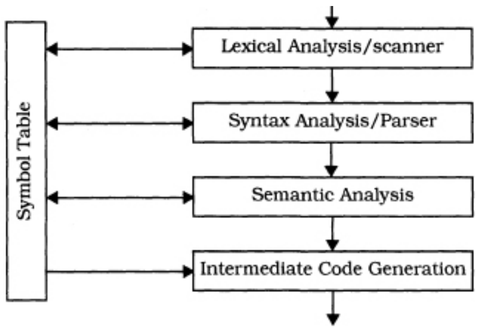
```
      $ Mocha <file.Mocha | optauLait | auLait2SPARC >a.out

      # interpret a Mocha program
      $ Mocha <file.Mocha | cup
```

`Mocha` is the frontend program, the optimizer is called `optauLait`, and the backends follow the naming convention `auLait2`*`machine`*. The interpreter is named cup after the virtual machine it implements.

Example Resolved

We did not follow the Pipes and Filters pattern strictly in our Mocha compiler by implementing all phases of the compiler as separate filter programs connected by pipes. We did this for performance reasons, and also because, in contrast to the third force, these phases do share a global state—the symbol table. It is sometimes possible to remove the need for shared global states by passing global information along the pipeline as additional data. However, this involves more complex data structures and an increase in pipeline data volume, imposing a performance penalty. Where the data being processed consists of simple types such as lines of text, such complex additional data structures have to be encoded and decoded by each filter.

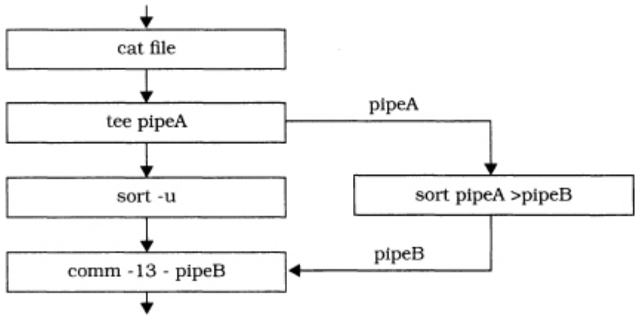


We combined the first four compiler phases into a single program because they all access and modify the symbol table. This allows us to implement the pipe connection between the scanner and the parser as a simple function call. Backends, interpreters or a debugger would also benefit from access to the symbol table. We therefore follow the example of many existing compilers by encoding some of the symbol table information, such as names and source line numbers, into the binary code for debugging purposes. Note that such symbol table information can greatly increase the size of compiled programs.

Variants

Tee and join pipeline systems. The single-input single-output filter specification of the Pipes and Filters pattern can be varied to allow filters with more than one input and/or more than one output. Processing can then be set up as a directed graph that can even contain feedback loops. The design of such a system, especially one with feedback loops, requires a solid foundation to explain and understand the complete calculation—a rigorous theoretical analysis and specification using formal methods are appropriate, to prove that the system terminates and produces the desired result. If we restrict ourselves to simple directed acyclic graphs, however, it is still possible to build useful systems. The UNIX filter program `tee`, for example, allows you to write data passed along a pipe to a file or to another 'named' pipe. Some filter programs allow the use of files or named pipes as input, as well as standard input. For example, to build a sorted list of all lines that occur more than once in a text file, we can construct the following shell program:

```
# first create two auxiliary named pipes to be used
mknod pipeA p
mknod pipeB p
# now do the processing using available UNIX filters
# start side fork of processing in background:
sort pipeA > pipeB &
# the main pipeline
cat file | tee pipeA | sort -u | comm -13 - pipeB
```



Known Uses

UNIX [Bac86] popularized the Pipes and Filters paradigm. The command shells and the availability of many filter programs made this approach to system development popular. As a system for software developers, frequent tasks such as program compilation and documentation creation are done by pipelines on a ‘traditional’ UNIX system. The flexibility of UNIX pipes made the operating system a suitable platform for the binary reuse of filter programs and for application integration.

CMS Pipelines [HRV95] is an extension to the operating system of IBM mainframes to support Pipes and Filters architectures. The implementation of CMS pipelines follows the conventions of CMS, and defines a *record* as the basic data type that can be passed along pipes, instead of a byte or ASCII character. CMS Pipelines provides a reuse and integration platform in the same way as UNIX. Because the CMS operating system does not use a uniform I/O-model in the same way as UNIX, CMS Pipelines defines *device drivers* that act as data sources or sinks, allowing the handling of specific I/O-devices within pipelines.

LASSPTools [Set95] is a toolset to support numerical analysis and graphics. The toolset consists mainly of filter programs that can be combined using UNIX pipes. It contains graphical input devices for analog input of numerical data using knobs or sliders, filters for numerical analysis and data extraction, and data sinks that produce animations from numerical data streams.

Consequences

The Pipes and Filters architectural pattern has the following **benefits**:

No intermediate files necessary, but possible. Computing results using separate programs is possible without pipes, by storing intermediate results in files. This approach clutters directories, and is error-prone if you have to set up your processing stages every time you run your system. In addition, it rules out incremental and parallel computation of results. Using Pipes and Filters removes the need for intermediate files, but allows you to investigate intermediate data by using a T-junction in your pipeline.

Flexibility by filter exchange. Filters have a simple interface that allows their easy exchange within a processing pipeline. Even if filter components call each other directly in a push or pull fashion, exchanging a filter component is still straightforward. In our compiler example a scanner generated with `lex` can easily be replaced by a more efficient hand-coded function `yylex()` that performs the same task. Filter exchange is generally not possible at run-time due to incremental computation in the pipeline.

Flexibility by recombination. This major benefit, combined with the reusability of filter components, allows you to create new processing pipelines by rearranging filters or by adding new ones. A pipeline without a data source or sink can be embedded as a filter within a larger pipeline. You should aim to tune the system platform or surrounding infrastructure to support this flexibility, such as is provided by the pipe mechanism and shells in UNIX.

Reuse of filter components. Support for recombination leads to easy reuse of filter components. Reuse is further enhanced if you implement each filter as an active component, while the underlying platform and shell allow easy end-user construction of pipelines.

Rapid prototyping of pipelines. The preceding benefits make it easy to prototype a data-processing system from existing filters. After you have implemented the principal system function using a pipeline you can optimize it incrementally. You can do this, for example, by developing specific filters for time-critical processing stages, or by re-implementing the pipeline using more efficient pipe connections. Your prototype pipeline can however be the final system if it performs the required task adequately. Highly-flexible filters such as the UNIX tools `sed` and `awk` reinforce such a prototyping approach.

Efficiency by parallel processing. It is possible to start active filter components in parallel in a multiprocessor system or a network. If each filter in a pipeline consumes and produces data incrementally they can perform their functions in parallel.

Applying the Pipes and Filters pattern imposes some **liabilities**:

Sharing state information is expensive or inflexible. If your processing stages need to share a large amount of global data, applying the Pipes and Filters pattern is either inefficient or does not provide the full benefits of the pattern.

Efficiency gain by parallel processing is often an illusion. This is for several reasons:

- The cost for transferring data between filters may be relatively high compared to the cost of the computation carried out by a single filter. This is especially true for small filter components or pipelines using network connections.
- Some filters consume all their input before producing any output, either because the task, such as sorting, requires it or because the filter is badly coded, for example by not using incremental processing when the application allows it.
- Context-switching between threads or processes is generally an expensive operation on a single-processor machine.
- Synchronization of filters via pipes may stop and start filters often, especially when a pipe has only a small buffer.

Data transformation overhead. Using a single data type for all filter input and output to achieve highest flexibility results in data conversion overheads. Consider a system that performs numeric calculations and uses UNIX pipes. Such a system must convert ASCII characters to real numbers, and vice-versa, within each filter. A simple filter, such as one that adds two numbers, will spend most of its processing time doing format conversion.

Error handling. As we explained in step 5 of the Implementation section, error handling is the Achilles' heel of the Pipes and Filters pattern. You should at least define a common strategy for error reporting and use it throughout your system. A concrete error-recovery or error-handling strategy depends on the task you need to solve. If your intended pipeline is used in a 'mission-critical' system and restarting the pipeline or ignoring errors is not possible, you should consider structuring your system using alternative architectures such as Layers (31).

See Also

The *Layers* pattern (31) is better suited to systems that require reliable operation, because it is easier to implement error handling than with Pipes and Filters. However, Layers lacks support for the easy recombination and reuse of components that is the key feature of the Pipes and Filter pattern.

Credits

This pattern relies on experience we gained when learning, using, and teaching UNIX. Our thanks therefore go to the designers of the first versions of UNIX, and its predecessors, who invented and established the use of pipes and filters.

The distinction of active and passive pipeline components was influenced by the PLoP'95 paper 'The Pipeline Design Pattern' [VBT95].

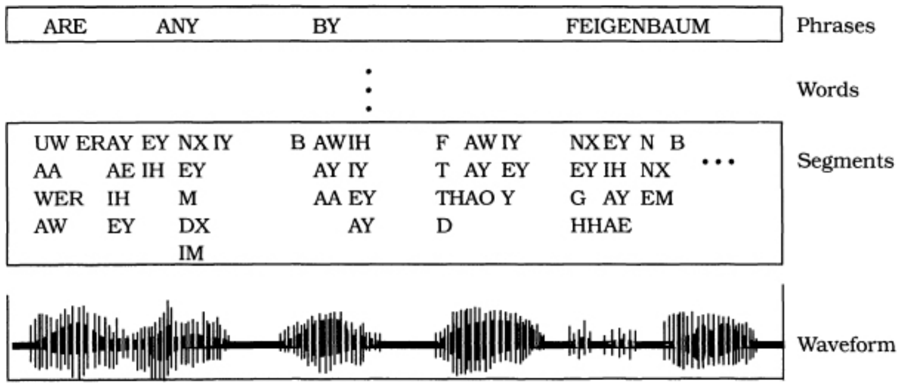
We also thank Ken Auer, Norbert Portner, Douglas C. Schmidt, Jiri Soukup, and John Vlissides for their valuable criticism and their suggestions for the improvement of the [PLoP94] version of this pattern.

Blackboard

The *Blackboard* architectural 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.

Example

Consider a software system for speech recognition. The input to the system is speech recorded as a waveform. The system not only accepts single words, but also whole sentences that are restricted to the syntax and vocabulary needed for a specific application, such as a database query. The desired output is a machine representation of the corresponding English phrases. The transformations involved require acoustic-phonetic, linguistic, and statistical expertise. For example, one procedure divides the waveform into segments that are meaningful in the context of speech, such as *phones*[6]. At the other end of the processing sequence, another procedure checks the syntax of candidate phrases. Both procedures work in different domains.



This diagram is mostly taken from [EHLR88]. The input is the waveform at the bottom, and the output consists of the phrase 'are any by Feigenbaum'.

For the moment we assume that there is no consistent algorithm that combines all the necessary procedures for recognizing speech—we discuss this topic further in the Consequences section. To make matters worse, the problem is characterized by the ambiguities of spoken language, noisy data, and the individual peculiarities of speakers such as vocabulary, pronunciation, and syntax.

Context

An immature domain in which no closed approach to a solution is known or feasible.

Problem

The Blackboard pattern tackles problems that do not have a feasible deterministic solution for the transformation of raw data into high-level data structures, such as diagrams, tables or English phrases. Vision, image recognition, speech recognition and surveillance are examples of domains in which such problems occur. They are characterized by a problem that, when decomposed into subproblems, spans several fields of expertise. The solutions to the partial problems require different representations and paradigms. In many cases no predetermined strategy exists for how the 'partial problem solvers' should combine their knowledge. This is in contrast to functional decomposition, in which several solution steps are arranged so that the sequence of their activation is hard-coded.

In some of the above problem domains you may also have to work with uncertain or approximate knowledge. Each transformation step can also generate several alternative solutions. In such cases it is often enough to find an optimal solution for most cases, and a suboptimal solution, or no solution, for the rest. The limitations of a Blackboard system therefore have to be documented carefully, and if important decisions depend on its results, the results have to be verified.

The following *forces* influence solutions to problems of this kind:

- A complete search of the solution space is not feasible in a reasonable time. For example, if you consider phrases of up to ten words using a vocabulary of a thousand words, the number of possible permutations of words is in the order of 1000^{10}.
- Since the domain is immature, you may need to experiment with different algorithms for the same subtask. For this reason, individual modules should be easily exchangeable.
- There are different algorithms that solve partial problems. For example, the detection of phonetic segments in the waveform is unrelated to the generation of phrases based on words and word sequences.

- Input, as well as intermediate and final results, have different representations, and the algorithms are implemented according to different paradigms.
- An algorithm usually works on the results of other algorithms.
- Uncertain data and approximate solutions are involved. For example, speech often includes pauses and extraneous sounds. These significantly distort the signal. The process of interpretation of the signal is also error-prone. Competing alternatives for a recognition target may occur at any stage of the process. For example, it is hard to distinguish between ‘till’ and ‘tell’. The words ‘two’ and ‘too’ even have the same pronunciation, as do many others in English.
- Employing disjoint algorithms induces potential parallelism. If possible you should avoid a strictly sequential solution.

Artificial Intelligence (AI) systems have been used with some success for such complex non-deterministic problems. In the ‘classical’ expert system structure, the input to the system and intermediate results are kept in working memory. The memory contents are used by an inference engine in conjunction with the knowledge base to infer new intermediate results. Such manipulation steps are repeated until some completion condition is fulfilled.

This type of expert system structure is inadequate for a speech recognition system. There are three reasons for this:

- All partial problems are solved using the same knowledge representation. However, the components involved in the speech recognition process work on fields of knowledge that differ as widely as the segmentation of a waveform and the parsing of candidate phrases. They therefore require different representations.
- The expert system structure provides only one inference engine to control the application of knowledge. Different partial problems with different representations require separate inference engines.
- In a ‘classical’ expert system, control is implicit in the structure of the knowledge base, for example in the ordering of the rules in a rule-based system. This is consistent with the view of many AI systems that ‘problem solving is search’ and ‘knowledge prunes and directs search’. This implies that you search in the search tree for nodes that include the solution for your problem, and use items of knowledge to guide your way from the root of the search tree—where all solutions are possible—to a single leaf.

  For the speech recognition problem, the view ‘problem solving is experts assembling their knowledge’ is more suitable. In other words, fragments of knowledge have to be applied at an opportune time, rather than in a predetermined order.

Solution

The idea behind the Blackboard architecture is a collection of independent programs that work cooperatively on a common data structure. Each program is specialized for solving a particular part of the overall task, and all programs work together on the solution. These specialized programs are independent of each other. They do not call each other, nor is there a predetermined sequence for their activation. Instead, the direction taken by the system is mainly determined by the current state of progress. A central control component evaluates the current state of processing and coordinates the specialized programs. This data-directed control regime is referred to as *opportunistic problem solving*. It makes experimentation with different algorithms possible, and allows experimentally-derived heuristics to control processing.

During the problem-solving process the system works with partial solutions that are combined, changed or rejected. Each of these solutions represents a partial problem and a certain stage of its solution. The set of all possible solutions is called the *solution space*, and is organized into levels of abstraction. The lowest level of solution consists of an internal representation of the input. Potential solutions of the overall system task are on the highest level.

The name ‘blackboard’ was chosen because it is reminiscent of the situation in which human experts sit in front of a real blackboard and work together to solve a problem. Each expert separately evaluates the current state of the solution, and may go up to the blackboard at any time and add, change or delete information. Humans usually decide themselves who has the next access to the blackboard. In the pattern we describe, a *moderator* component decides the order in which programs execute if more than one can make a contribution.

Structure

Divide your system into a component called *blackboard*, a collection of *knowledge sources*, and a *control* component.

The *blackboard* is the central data store. Elements of the solution space and control data are stored here. We use the term *vocabulary* for the set of all data elements that can appear on the blackboard. The blackboard provides an interface that enables all knowledge sources to read from and write to it.

All elements of the solution space can appear on the blackboard. For solutions that are constructed during the problem solving process and put on the blackboard, we use the terms *hypothesis* or *blackboard entry*. Hypotheses rejected later in the process are removed from the blackboard.

A hypothesis usually has several attributes, for example its *abstraction level* that is, its conceptual distance from the input. Hypotheses that have a low abstraction level have a representation that is still similar to input data representation, while hypotheses with the highest abstraction level are on the same abstraction level as the output. Other hypothesis attributes are the estimated degree of truth of the hypothesis or the time interval covered by the hypothesis.

It is often useful to specify relationships between hypotheses, such as ‘part-of’ or ‘in-support-of’.

➥ The solution space for the speech recognition example consists of acoustic-phonetic and linguistic speech fragments. The levels of abstraction are signal parameters, acoustic-phonetic segments, phones, syllables, words, and phrases.

The degree of truth for a syllable is estimated by the quality of the match between the ideal phone sequences for that syllable and the hypothesized phones.

The waveform of the acoustic signal is recorded on a time axis that corresponds to the X-axis in the figure on page 71. Every solution has an attribute that specifies the interval on the X-axis that it describes.

The blackboard can be viewed as a three-dimensional problem space with the time line for speech on the X-axis, increasing levels of abstraction on the Y-axis and alternative solutions on the Z-axis [Nii86].

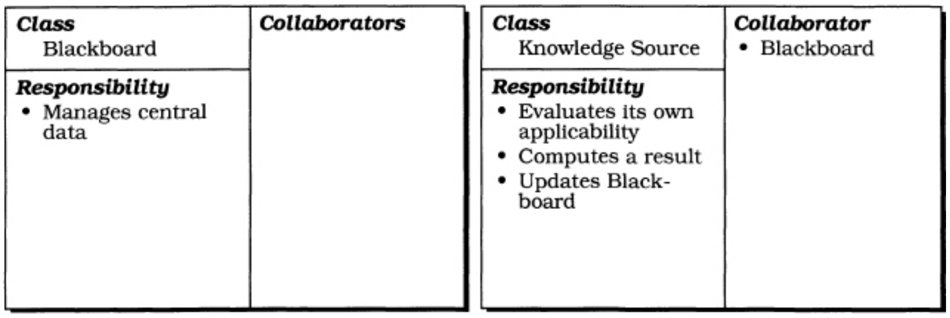
Knowledge sources are separate, independent subsystems that solve specific aspects of the overall problem. Together they model the overall problem domain. None of them can solve the task of the system alone—a solution can only be built by integrating the results of several knowledge sources.

➥ In the speech recognition system we specify solutions for the following partial problems: defining acoustic-phonetic segments, and creating phones, syllables, words and phrases. For each of these partial problem we define one or several knowledge sources. One knowledge source at the word level, for example, may create words from adjacent syllables, while another source on the same level verifies words that depend on neighboring words.

Note that the transformation from waveform to phrase is not necessarily a strictly sequential process. The complete waveform is not necessarily first transformed into segments, all segments into phones, then into syllables and words, and phrases then built. A portion of the waveform may have been transformed into words, another may have been rejected at the word level back to the phone level, and a third may not be analyzed at all until enough evidence on the phrase level exists to tackle it.

Knowledge sources do not communicate directly—they only read from and write to the blackboard. They therefore have to understand the vocabulary of the blackboard. We explore the ramifications of this in the Implementation section.

Often a knowledge source operates on two levels of abstraction. If a knowledge source implements forward reasoning, a particular solution is transformed to a higher-level solution. A knowledge source that reasons backwards searches at a lower level for support for a solution, and may refer it back to a lower level if the reasoning did not give support for the solution.



Class Blackboard	***Collaborators***
Responsibility • Manages central data	

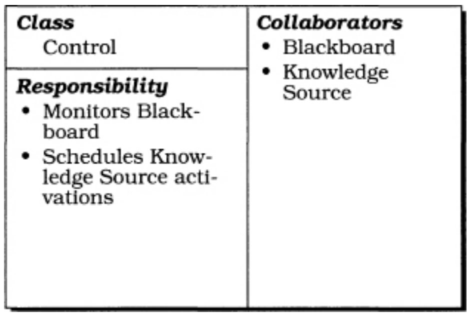
Class Knowledge Source	***Collaborator***
Responsibility • Evaluates its own applicability • Computes a result • Updates Blackboard	• Blackboard

Each knowledge source is responsible for knowing the conditions under which it can contribute to a solution. Knowledge sources are therefore split into a *condition-part* and an *action-part.* The condition-part evaluates the current state of the solution process, as written on the blackboard, to determine if it can make a contribution. The action-part produces a result that may cause a change to the blackboard's contents.

➥ Our speech recognition system has diverse knowledge sources that transform several hypotheses at the same level and with contiguous time intervals to a single hypothesis on the next higher level. For example, a phrase is built from a selection of words that together span the time interval corresponding to the phrase. Other knowledge sources predict new hypotheses at the same level. For example, one knowledge source predicts possible words that might syntactically precede or follow a given phrase. We also define a knowledge source that verifies the predicted hypotheses based on information at the next lower level. This calculates the consistency between a predicted word and the set of segments that span the same time interval.

The *control* component runs a loop that monitors the changes on the blackboard and decides what action to take next. It schedules knowledge source evaluations and activations according to a knowledge application *strategy*. The basis for this strategy is the data on the blackboard.

The strategy may rely on *control knowledge sources*. These special knowledge sources do not contribute directly to solutions on the blackboard, but perform calculations on which control decisions are made. Typical tasks are the estimation of the potential for progress, or the computational costs for execution of knowledge sources. Their results are called *control data* and are put on the blackboard as well.



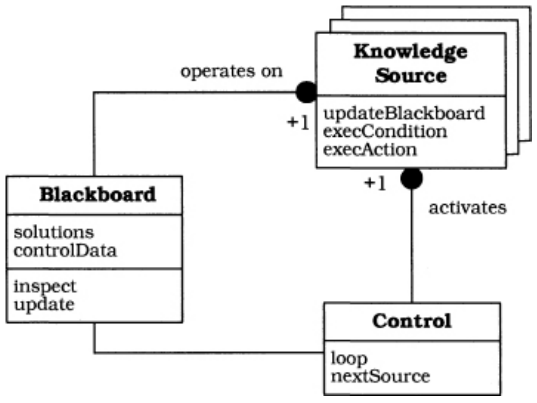
Class Control	***Collaborators***
Responsibility	• Blackboard
• Monitors Blackboard	• Knowledge Source
• Schedules Knowledge Source activations	

Theoretically, it is possible that the blackboard can reach a state at which no knowledge source is applicable. In this case, the system fails to deliver a result. In practice, it is more likely that each reasoning step introduces several new hypotheses, and that the number of possible next steps 'explodes'. The problem is therefore to restrict the alternatives to be taken rather than to find an applicable knowledge source.

A special knowledge source or a procedure in the control component determines when the system should halt, and what the final result is. The system halts when an acceptable[7] hypothesis is found, or when the space or time resources of the system are exhausted.

The following figure illustrates the relationship between the three components of the Blackboard architecture. The blackboard component defines two procedures: `inspect` and `update`. Knowledge sources call `inspect` to check the current solutions on the blackboard, `update` is used to make changes to the data on the blackboard.

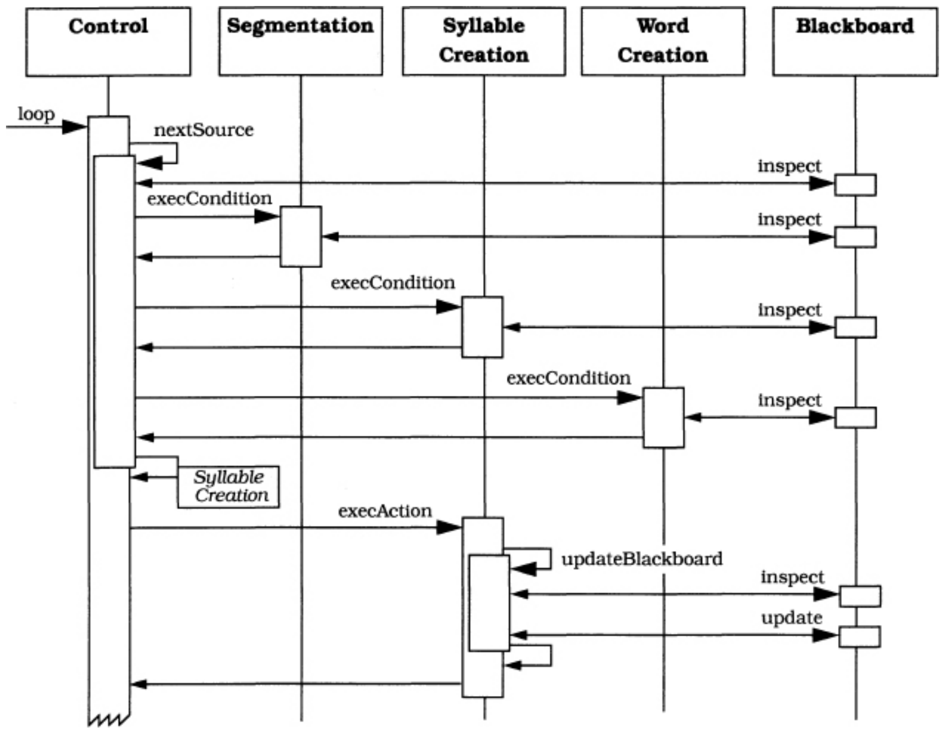
The Control component runs a loop that monitors changes on the blackboard and decides what actions to take next. We call the procedure responsible for this decision `nextSource ()`.



Dynamics

The following scenario illustrates the behavior of the Blackboard architecture. It is based on our speech recognition example:

- The main loop of the Control component is started.
- Control calls the `nextSource()` procedure to select the next knowledge source.
- `nextSource()` first determines which knowledge sources are potential contributors by observing the blackboard. In this example we assume the candidate knowledge sources are Segmentation, Syllable Creation and Word Creation.
- `nextSource()` invokes the condition-part of each candidate knowledge source. In the example, the condition-parts of Segmentation, Syllable Creation and Word Creation inspect the blackboard to determine if and how they can contribute to the current state of the solution.
- The Control component chooses a knowledge source to invoke, and a hypothesis or a set of hypotheses to be worked on. In the example the choice is made according to the results of the condition parts. In other cases the selection is also based on control data. It applies the action-part of the knowledge source to the hypotheses. In our speech recognition example, assume that Syllable Creation is the most promising knowledge source. The action-part of Syllable Creation inspects the state of the blackboard, creates a new syllable and updates the blackboard.



Implementation

To implement the Blackboard pattern, carry out the following steps:

1 *Define the problem:*

- Specify the domain of the problem and the general fields of knowledge necessary to find a solution.
- Scrutinize the input to the system. Determine any special properties of the input such as noise content or variations on a theme—that is, does the input contain regular patterns that change slowly over time?
- Define the output of the system. Specify the requirements for correctness and fail-safe behavior. If you need an estimation of the credibility of the results, or if there are cases in which the system should ask

the user for further resources, record this.
- Detail how the user interacts with the system.

➥ The fields of knowledge important for a system in the domain of speech recognition are acoustics, linguistics and statistics. The input is a sequence of acoustic signals from a speaker. The data is noisy. If the system allows the speaker to repeat a phrase several times, the input contains 'variations on a theme', as described above. The desired output is a written English phrase corresponding to the spoken phrase. When used for a database query interface, the system can tolerate occasional misinterpretations. If we have to repeat a query in, say, 10% of cases, the system can still be useful.
2 *Define the solution space for the problem.* We distinguish intermediate and top-level solutions on one hand, and partial and complete solutions on the other. A *top-level* solution is at the highest abstraction level. Solutions at other levels are *intermediate* solutions. A *complete* solution solves the whole problem, whereas a *partial* solution solves part of the problem. Note that complete solutions can belong to intermediate levels, and a partial solution may be top-level.
➥ In speech recognition, complete top-level solutions are phrases that are correct with respect to a defined vocabulary and syntax. Complete intermediate solutions are sequences of acoustic-phonetic or linguistic elements that describe the whole spoken phrase. Parts of solutions are the elements themselves.
So, perform the following steps:

- Specify exactly what constitutes a top-level solution.
- List the different abstraction levels of solutions.
- Organize solutions into one or more abstraction hierarchies.
- Find subdivisions of complete solutions that can be worked on independently, for example words of a phrase or regions of a picture or area.

3 *Divide the solution process into steps:*

- Define how solutions are transformed into higher-level solutions.
- Describe how to predict hypotheses at the same abstraction level.
- Detail how to verify predicted hypotheses by finding support for them in other levels.
- Specify the kind of knowledge that can be used to exclude parts of the solution space.

➥ To transform solutions on the syllabic level to solutions on the word level, we provide a dictionary that associates a syllable with all the words whose pronunciation contains the syllable.
Syntactic and statistical knowledge is useful when pruning the search for word sequences. For example, the heuristic that an adjective is normally followed by another adjective or a noun can be used to cut down computing time.
4 *Divide the knowledge into specialized knowledge sources with certain subtasks.* These subtasks often correspond to areas of specialization. There may be some subtasks for which the system defers to human specialists for decisions about dubious cases, or even to replace a missing knowledge source. Knowledge sources must be complete in the following sense: for most of the input phrases, at least one possible sequence of knowledge source activations that leads to an acceptable solution should exist.
➥ Examples of knowledge sources are segmentation, phone creation, syllable creation, word creation, phrase creation, word prediction and word verification.
5 *Define the vocabulary of the blackboard.* Elaborate your first definition of the solution space and the abstraction levels of your solutions. Find a representation for solutions that allows all knowledge sources to read from and contribute to the blackboard. This does not mean that each knowledge source must understand every blackboard entry, but each knowledge source must be able to decide whether it can use a blackboard entry. If necessary, provide components that translate between blackboard entries and the internal representations within knowledge sources. This allows knowledge sources to be easily exchanged, to be independent of each other's representation and paradigms, and at the same time use each other's results.
➥ In our speech recognition example, each hypothesis has a uniform attribute-value structure. Some

attributes must be included in all hypotheses, while others are optional. The element name, the abstraction level and the time interval covered by the hypothesis are among the required attributes. The estimated degree of truth is optional. For example, the blackboard may contain the following entry:

```
ABOUT+FEIGENBAUM+AND+FELDMAN+] (phrase) (48:225) (83).
```

Depending on the abstraction level, each knowledge source can decide if it is able to work on a hypothesis or not. The knowledge source responsible for segmentation, for example, does not understand the symbols '+' and ']' in the blackboard entry shown here. It knows, by reading the value of the attribute abstraction level, that the hypothesis is a phrase, so it does not check the other attributes.

To evaluate the contents of the blackboard, the Control component must be able to understand it. The vocabulary of the blackboard cannot therefore be defined once, but evolves in concert with the definition of knowledge sources and the Control component. At some point during design the vocabulary must stabilize, to allow the development of stable interfaces to the knowledge sources.

6 *Specify the control of the system*. The Control component implements an opportunistic problem-solving strategy that determines which knowledge sources are allowed to make changes to the blackboard. The aim of this strategy is to construct a hypothesis that is acceptable as a result. But when is a hypothesis acceptable? Since the correctness of a hypothesis is not verifiable in a strict sense, our goal is to construct the most credible complete, top-level solution possible in the solution space.

The *credibility* of a hypothesis is the likelihood that it is correct. We estimate the credibility of a hypothesis by considering all plausible alternatives to it, and the degree of support each alternative receives from the input data. The credibility rating is, for example, a number on a scale ranging from 0 to 100. A hypothesis is acceptable if it is top-level and complete and if its assessed credibility reaches a threshold value, for example 85. To find an acceptable hypothesis, the system eliminates hypotheses with a low credibility, and detects mutually-supportive clusters of hypotheses that are consistent with the input data.

In the simplest case the control strategy consults the condition-part of all knowledge sources whenever the blackboard is changed, and picks one of the applicable knowledge sources for activation at random. However, this strategy usually is too inefficient, as progress toward an acceptable hypothesis is slow. The design of a good control strategy is the most difficult part of the system design. It often consists of a tedious process of trying combinations of several mechanisms and partial strategies. The Strategy pattern [GHJV95] is useful here to support an exchange of control strategies, even at run-time. Sophisticated control strategies may be implemented by a dedicated knowledge-based system.

The following mechanisms optimize the evaluation of knowledge sources, and so increase the effectiveness and performance of the control strategy:

- Classifying changes to the blackboard into two types. One type specifies all blackboard changes that may imply a new set of applicable knowledge sources, the other specifies all blackboard changes that do not. After changes of the second type, the Control component chooses a knowledge source without another invocation of all condition-parts.
- Associating categories of blackboard changes with sets of possibly applicable knowledge sources.
- Focusing of control. The *focus* contains either partial results on the blackboard that should be worked on next, or knowledge sources that should be preferred over others.
- Creating a queue in which knowledge sources classified as applicable wait for their execution. By using a queue, you save valuable information about knowledge sources rather than discarding it after each change to the blackboard.

Control strategies use *heuristics* to determine which of the applicable knowledge sources to activate. Heuristics are rules based on experience and guesses. Keep in mind that good heuristics work often, but not always. Here are some examples of heuristics that can be used by control strategies:

- Prioritizing applicable knowledge sources. The basis for such a priority calculation is the evaluation of the condition-parts of knowledge sources, and possibly other information such as the potential for making progress using a knowledge source, and the costs of its application. The Control component may consider the contributions of knowledge-sources to decide about prioritization. In this case it must execute the action-parts of all applicable knowledge sources before it can decide which should make a change to the blackboard. If the system uses a queue, the priority of each knowledge source is stored with its entry. A change to the blackboard may result in a change in priorities or the removal of knowledge sources from the queue.

- Preferring low-level or high-level hypotheses. If this is the only strategy used, the control strategy is no longer opportunistic, but rather implements forward- or backward-chaining.
- Preferring hypotheses that cover large parts of the problem.
- 'Island driving'. This strategy involves assuming that a particular hypothesis is part of an acceptable solution, and is considered as an 'island of certainty'. Knowledge source activations that work on this hypothesis are then preferred over others, which removes the need to search constantly for alternative hypotheses with higher priorities.

If the control component displays complex and independent subtasks, define one control knowledge source for each of these subtasks. Treat them like other knowledge sources. For example, the priority calculation for applicable knowledge sources can itself be implemented as a dedicated control knowledge source.
7 *Implement the knowledge sources*. Split the knowledge sources into condition-parts and action-parts according to the needs of the Control component. To maintain the independency and exchangeability of knowledge sources, do not make any assumptions about other knowledge sources or the Control component.
You can implement different knowledge sources in the same system using different technologies. For example, one may be a rule-based system, another a neural net and a third a set of conventional functions. This implies that the knowledge sources themselves may be organized according to diverse architectural or design patterns. For example, one knowledge source may be designed using the Layers pattern (31), while another may be structured according to the Reflection pattern (193). If you intend to develop your system using object-oriented technology, but your knowledge sources are implemented using another paradigm, it makes sense to 'wrap' them using the Facade pattern [GHJV95].

Variants

Production System. This architecture is used in the OPS language [FMcD77]. In this variant subroutines are represented as condition-action rules, and data is globally available in working memory. Condition-action rules consist of a left-hand side that specifies a condition, and a right-hand side that specifies an action. The action is executed only if the condition is satisfied and the rule is selected. The selection is made by a 'conflict resolution module'. A Blackboard system can be regarded as a radical extension of the original production system formalism: arbitrary programs are allowed for both sides of the rules, and the internal complexity of the working memory is increased. Complicated scheduling algorithms are used for conflict-resolution.

Repository. This variant is a generalization of the Blackboard pattern. The central data structure of this variant is called a *repository*. In a Blackboard architecture the current state of the central data structure, in conjunction with the Control component, finally activates knowledge sources. In contrast, the Repository pattern does not specify an internal control. A repository architecture may be controlled by user input or by an external program. A traditional database, for example, can be considered as a repository. Application programs working on the database correspond to the knowledge sources in the Blackboard architecture.

Examples of repository systems that are not Blackboard systems are given in [SG96]: 'Programming environments are often organized as a collection of tools together with a shared repository of programs and program fragments. Even applications that have been traditionally viewed as pipeline architectures, may be more accurately interpreted as repository systems…’ Compilers, for example, have traditionally been described and sometimes also been implemented as pipelines[8]. Modern compilers have a repository that holds shared information such as symbol tables and abstract syntax trees. The compilation phases correspond to knowledge sources operating on the repository. This architecture enables incremental problem solving:

- The *scanner* reads an identifier that is not yet defined.
- The *parser* recognizes the syntactical unit described by the identifier.
- The *code generator* then jumps in and creates the corresponding machine code, if any.

Known Uses

HEARSAY-II. The first Blackboard system was the HEARSAY-II speech recognition system from the early 1970’s. It was developed as a natural language interface to a literature database. Its task was to answer queries about documents and to retrieve documents from a collection of abstracts of Artificial Intelligence publications. The inputs to the system were acoustic signals that were semantically interpreted and then transformed to a database query. [EM88] gives a detailed description and retrospective view of the project. Selected aspects of HEARSAY-II also serve as the running example of this pattern. The following paragraphs discuss its control aspects.

In HEARSAY-II, the condition-part of a knowledge source identifies a configuration of hypotheses on the blackboard appropriate for action by the knowledge source. This subset is called the *stimulus frame*. For example, the condition-part of the knowledge source that generates phrase hypotheses looks for contiguous word or phrase hypotheses. Condition-parts also calculate a formal description of the likely action that the knowledge source will perform, called the *response frame*. For example, a response frame for a word hypothesizer based on syllables indicates that its action will be to generate hypotheses at the word level, and that the interval covered by the hypothesis on the X-axis will include at least the stimulus frame.

The control component of HEARSAY-II consists of the following:

- The *focus-of-control database*, which contains a table of *primitive change types* of blackboard changes, and those condition-parts that can be executed for each change type. Examples of primitive change types are ‘new syllable’ or ‘new word created bottom-up’—indicating that a new word appeared on the blackboard and it was inferred using hypotheses on lower levels.
- The *scheduling queue*, which contains pointers to condition- or action-parts of knowledge sources.[9]
- The *monitor*, which keeps track of each change made to the black board. The monitor inserts pointers to applicable condition-parts into the scheduling queue based on the corresponding primitive change types. If a condition-part is actually executed and the calculated response frame is not empty, a pointer to the matching action-part is placed in the scheduling queue.
- The *scheduler*, which uses experimentally-derived heuristics to calculate priorities for the condition- and action-parts waiting in the scheduling queue. This estimation is based on the specific stimulus and response frames. It also takes into account overall blackboard state information, such as which out of several competing hypotheses in the same X-axis interval has highest support from hypotheses on lower levels. The scheduler finally selects the condition- or action-part with the highest priority for execution [LeEr88].

The designers of HEARSAY-II combined several problem-solving techniques for their knowledge application strategy. The first is a bottom-up approach in which interpretations are synthesized directly from the data, working up the abstraction hierarchy. The second is a top-down strategy, in which hypotheses at lower levels are produced recursively until a sequence of hypotheses on the lowest level is produced that can be tested against the original input. Orthogonal to those approaches, HEARSAY-II employs a 'generate-and-test' strategy, in which a knowledge source generates hypotheses, and their validity is evaluated by another knowledge source.

HASP/SIAP. The HASP system was designed to detect enemy submarines. In this system, hydrophone arrays monitor a sea area by collecting sonar signals. A Blackboard system interprets these signals [Nii86]. HASP is an event-based system in the sense that the occurrence of a particular event implies that new information is available. The blackboard is used as a 'situation board' that evolves over time. Since information is collected continuously, there is information redundancy as well as new and different information. HASP deals with multiple input streams. Besides the low-level data from hydrophones, it accepts high-level descriptions of the situation gathered from intelligence or other sources.

CRYSALIS. This system was designed to infer the three-dimensional structure of protein molecules from X-ray diffraction data [Ter88]. The system introduces several features to the Blackboard architecture. The blackboard is divided into several parts called *panels*. Each panel has its own vocabulary and hierarchy. It is possible to restrict access to certain panels by knowledge sources. CRYSALIS uses a data panel and a hypothesis panel. Knowledge sources are organized into levels. Only the lowest level contains knowledge sources that actually create and modify hypotheses. The other levels consist of control knowledge sources. CRYSALIS was the first Blackboard system to use rule-based systems for control.

TRICERO. This system monitors aircraft activities. It extends the Blackboard architecture to distributed computing [Wil84]. Four complete, independent expert systems for partial problems were designed to run on four separate machines.

Generalizations. Between 1977 and 1984 application-oriented Blackboard systems were generalized to produce frameworks intended to ease building Blackboard applications. However, no standard way to do this emerged.

SUS. A recent project called 'Software Understanding System', described in [THG94], is particularly interesting from our point of view as software pattern authors. The aim of SUS is to support understanding of software, and the search for reusable assets. In a matching process the system compares patterns from a pattern base to the system under analysis. SUS incrementally builds a 'pattern map' of the analyzed software that then can be viewed.

Example Resolved

In the following we present an excerpt of the processing steps that HEARSAY-II performs to understand the phrase 'Are any by Feigenbaum and Feldman?', as described in [EHLR88].

To briefly characterize the knowledge sources that are activated in the example:

- `RPOL` runs as a high-priority task immediately after any knowledge source activity that creates a new hypothesis. `RPOL` uses rating information on the new hypothesis, as well as rating information on hypotheses to which the new hypothesis is connected, to calculate an overall rating for the new hypothesis.

- `PREDICT` works on a phrase and generates predictions of all words that can immediately precede or follow the phrase in the language.
- `VERIFY` tries to verify the existence of, or reject, a predicted word, in the context of the phrase that predicts it. If the word is verified a confidence rating must also be generated for it. This is done by the knowledge source RPOL.
- `CONCAT` accomplishes the generation of a phrase from a verified word and its predicting phrase. The extended phrase includes a rating that is based on the ratings of the predicting phrase, and the verified word. If a verified word is already associated with some other phrase, `CONCAT` tries to parse that phrase with the predicting phrase. If successful, a phrase hypothesis is created which represents the merging of the two phrases.

We have simplified the original description for ease of understanding, and have omitted explicit executions of the condition-parts of knowledge sources. Executions of `RPOL` are also omitted. An execution of the `VERIFY` knowledge source often immediately follows the execution of the `PREDICT` knowledge source. The two knowledge source executions are therefore combined into one step.

To help you understand the following sequence of processing steps and the figure, here is an explanation of the notation we have used:

- The number in brackets behind a word or phrase denotes its credibility rating.
- '[' marks the begin of an spoken phrase, and ']' marks its end.
- KS stands for 'knowledge source'.

The highest rated hypotheses on the blackboard are currently:

`[ARE+(97)`, `[ARE+REDDY(91)`, `FEIGENBAUM+AND+FELDMAN+] (85)`, and `[ARE+ANY(86)`.

Step 17 is the first step to consider:

```
Step 17: KS PREDICT&VERIFY
Stimulus: FEIGENBAUM+AND+FELDMAN+] (85) (phrase).
Action: Predict eight preceding words;
reject one: DISCUSS;
find three already on the blackboard: CITE(70),
ABOUT(75), BY(80);
verify four: CITES(65), QUOTE(70), ED(75), NOT(75).
```

The rating of a hypothesis is not the only parameter the Scheduler uses to assign priorities to waiting knowledge source activations. In particular, the length of a hypothesis is also important. The phrase `FEIGENBAUM+AND+FELDMAN+]` with a rating of 85 was therefore preferred over the phrases `[ARE+REDDY` with a rating of 91 and `[ARE+ANY` with a rating of 86, because it is much longer.

In steps 18 through 24, alternative word extensions of `FEIGENBAUM+AND+FELDMAN+]` (85) are explored. As a result of this exploration, the phrase `BY+FEIGENBAUM+AND+FELDMAN+]` (84) is considered the most credible.

```
Step 18: KS CONCAT
Stimulus: BY(80) (word),
FEIGENBAUM+AND+FELDMAN+] (85) (phrase).
Action: Create phrase:
```

```
BY+FEIGENBAUM+AND+FELDMAN+] (84).

Step 19: KS CONCAT
Stimulus: ABOUT(75) (word),
FEIGENBAUM+AND+FELDMAN+ (85) (phrase).
Action: Create phrase:
ABOUT+FEIGENBAUM+AND+FELDMAN+] (83)

Step 20: KS PREDICT&VERIFY
Stimulus:
ABOUT+FEIGENBAUM+AND+FELDMAN+] (83) (phrase).
Action: Predict one preceding word;
verify: WHAT(10).

Step 21: KS CONCAT
Stimulus: CITE(70) (word),
FEIGENBAUM+AND+FELDMAN+] (85) (phrase).
Action: Create phrase:
CITE+FEIGENBAUM+AND+FELDMAN+] (83) (phrase).

Step 22: KS PREDICT&VERIFY
Stimulus: CITE+FEIGENBAUM+AND+FELDMAN+] (83) (phrase).
Action: Predict ten preceding words;
reject five: ABSTRACTS, ARE, BOOKS, PAPERS, REFERENCED;
find two already on the blackboard: ANY(65), THESE(25);
verify three: ARTICLE(25), WRITTEN(25), ARTICLES(10).
```

If all ten word predictions preceding the phrase had been rejected, the phrase hypothesis itself would also be rejected.

```
Step 24: KS CONCAT
Stimulus: NOT(75) (word),
FEIGENBAUM+AND+FELDMAN+] (85).
Action: Create phrase:
NOT+FEIGENBAUM+AND+FELDMAN+] (83).

Step 25 KS CONCAT
Stimulus: ANY(65) (word),
BY+FEIGENBAUM+AND+FELDMAN+] (84) (phrase).
Action: Create phrase:
ANY+BY+FEIGENBAUM+AND+FELDMAN+] (82).
[ARE+ANY+BY+FEIGENBAUM+AND+FELDMAN+] (85)
is also created from [ARE+ANY(86) and
BY+FEIGENBAUM+AND+FELDMAN+] (84).
```

The phrase happens to be a complete sentence, and is therefore a candidate for the interpretation of the spoken input.

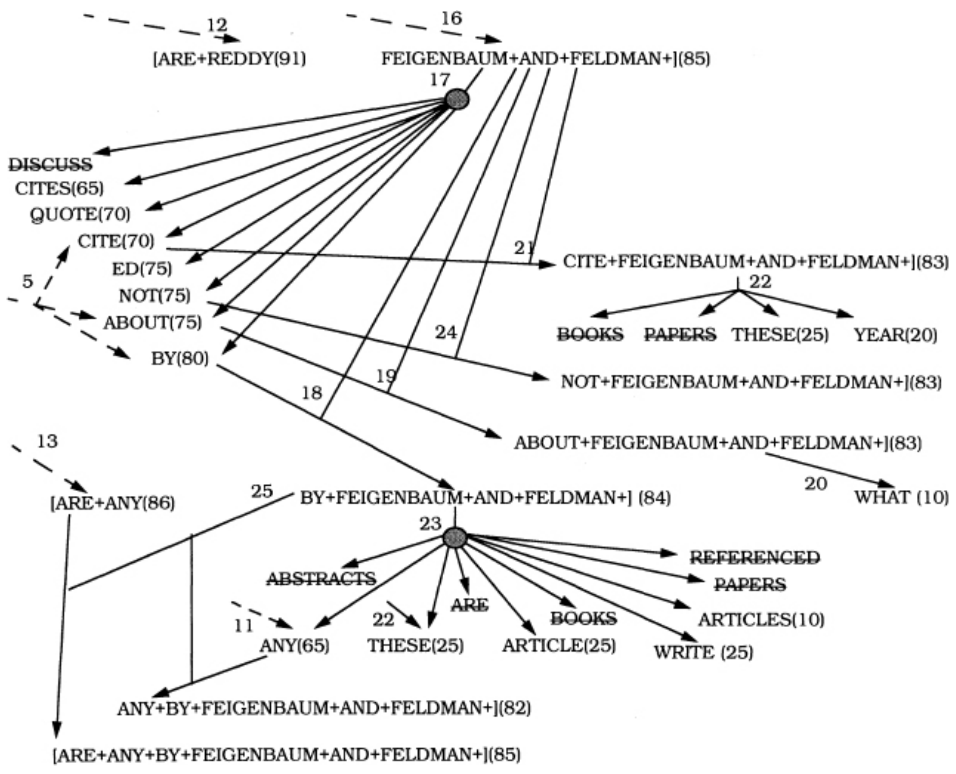
In the figure that follows, an arc points from one hypothesis to another if one hypothesis is derived from the other in a single processing step. The arc is labeled with the number of the processing step. Dashed arcs point to hypotheses that were already on the blackboard before step 17.

Consequences

The Blackboard approach to problem decomposition and knowledge application helps to resolve most of the forces listed in the problem section:

Experimentation. In domains in which no closed approach exists and a complete search of the solution space is not feasible, the Blackboard pattern makes experimentation with different algorithms possible, and also allows different control heuristics to be tried.

Support for changeability and maintainability. The Blackboard architecture supports changeability and maintainability because the individual knowledge sources, the control algorithm and the central data structure are strictly separated. However, all modules can communicate via the blackboard.



Reusable knowledge sources. Knowledge sources are independent specialists for certain tasks. A Blackboard architecture helps in making them reusable. The prerequisites for reuse are that knowledge source and the underlying Blackboard system understand the same protocol and data, or are close enough in this respect not to rule out adaptors for protocol or data.

Support for fault tolerance and robustness. In a Blackboard architecture all results are just hypotheses. Only those that are strongly supported by data and other hypotheses survive. This provides tolerance of noisy data and uncertain conclusions.

The Blackboard pattern has some **liabilities**:

Difficulty of testing. Since the computations of a Blackboard system do not follow a deterministic algorithm, its results are often not reproducible. In addition, wrong hypotheses are part of the solution process.

No good solution is guaranteed. Usually Blackboard systems can solve only a certain percentage of their given tasks correctly.

Difficulty of establishing a good control strategy. The control strategy cannot be designed in a straightforward way, and requires an experimental approach.

Low Efficiency. Blackboard systems suffer from computational overheads in rejecting wrong hypotheses. If no deterministic algorithm exists, however, low efficiency is the lesser of two evils when compared to no system at all.

High development effort. Most Blackboard systems take years to evolve. We attribute this to the ill-structured problem domains and extensive trial-and-error programming when defining vocabulary, control strategies and knowledge sources.

No support for parallelism. The Blackboard architecture does not prevent the use of a control strategy that exploits the potential parallelism of knowledge sources. It does not however provide for their parallel execution. Concurrent access to the central data on the blackboard must also be synchronized.

To summarize, the Blackboard architecture allows an interpretative use of knowledge. It evaluates alternative actions, chooses the best for the current situation, and then applies the most promising knowledge source. The expense for such deliberation can be justified so long as no adequate explicit algorithm is available for the problem. When such an algorithm emerges, it usually provides higher performance and effectiveness. The Blackboard architecture consequently lends itself best to immature domains in which experimentation is helpful. After research and the gaining of experience, better algorithms may evolve that allow you to use a more efficient architecture.

This occurred in the domain of speech recognition. For example, in the HARPY system, a successor to HEARSAY-II, most of the knowledge is precompiled into a unified structure that represents all possible spoken phrases [EHLR88]. All inter-level substitutions, such as segment to phone, phone to word, and word to phrase are compiled into a single enormous finite-state Markov network. An interpreter then compares segments of the spoken phrase with this structure to find a network path that most closely approximates the segmented speech signal. The search technique used, called *beam search*, combined with word lattices, is a heuristic form of dynamic programming. Acoustical and linguistic knowledge are no longer combined via a blackboard, but rather by a ‘maximum likelihood’ computation. A window slides over the input and continuously appends new results to the output. This allows speech recognition to be used in a real-time fashion. [Mar95] gives a recent update on simpler speech recognition products. For more details on speech recognition, see [HAJ90], [Rab86] and [Rab89].

Credits

Our Blackboard pattern is based mostly on features abstracted from the HEARSAY-II speech recognition system. We found the first reference to the term ‘blackboard’ in AI literature, in a text by Newell and Simon [NS72] concerned with the organizational problems of checkers-playing, chess-playing and theorem-proving programs. The most comprehensive descriptions and discussions of Blackboard systems are in [EM88] and [Cra95].

We thank Harald Höge from the Siemens speech processing group for explaining recent progress in this domain.

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:

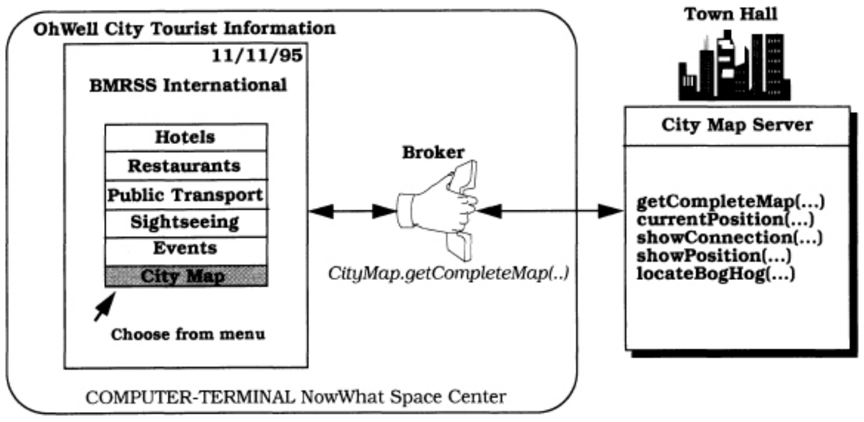
- Those working with an existing Broker system who are interested in understanding the architecture of such systems.
- Those who want to build 'lean' versions of a Broker system, without all the bells and whistles of a full-blown OLE or CORBA.
- Those who plan to implement a fully-fledged Broker system, 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 inter-operating 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 group semantically-related functionality. There are two kinds of servers:

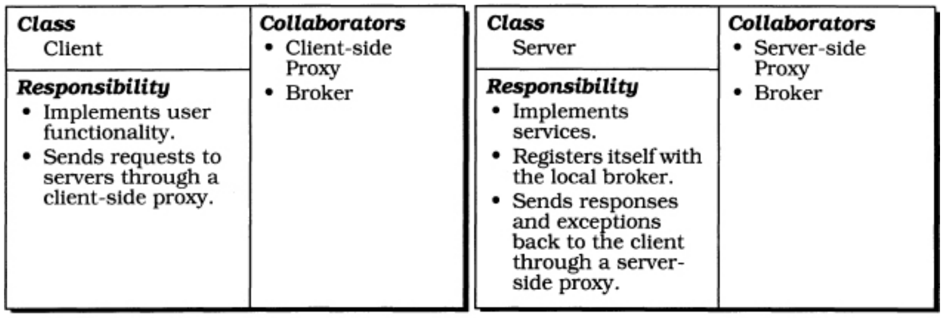
- Servers offering common services to many application domains.
- Servers implementing specific functionality for a single application domain or task.

➥ 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.



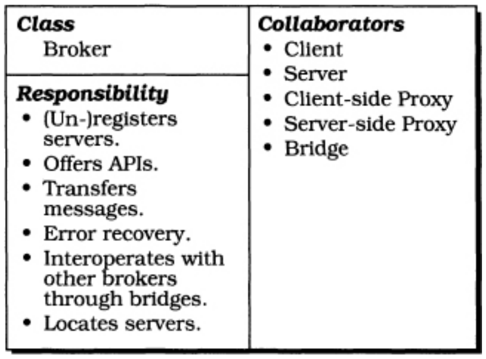
Class Client	**Collaborators**
Responsibility • Implements user functionality. • Sends requests to servers through a client-side proxy.	• Client-side Proxy • Broker

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

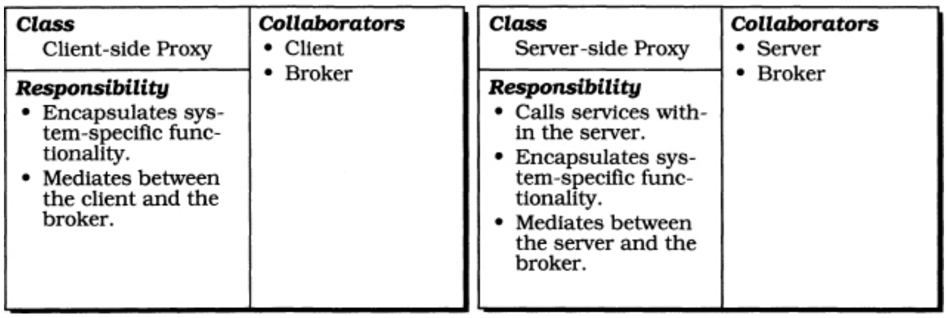
➥ A broker in our CIS example is the combination of an Internet gateway and the Internet infrastructure itself. Every information exchange between a client and a server must pass through the broker. A client specifies the information it wants using unique identifiers called URLs (Universal Resource Locators). 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:

- 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 • Encapsulates system-specific functionality. • Mediates between the client and the broker.	• Client • Broker

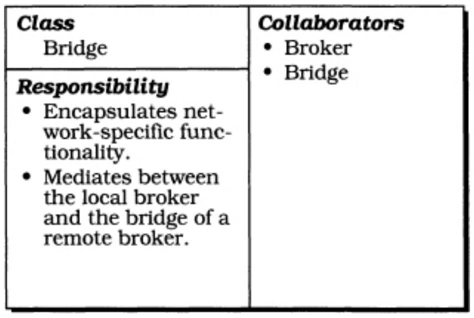
Class Server-side Proxy	**Collaborators**
Responsibility • Calls services within the server. • Encapsulates system-specific functionality. • Mediates between the server and the broker.	• Server • Broker

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 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).

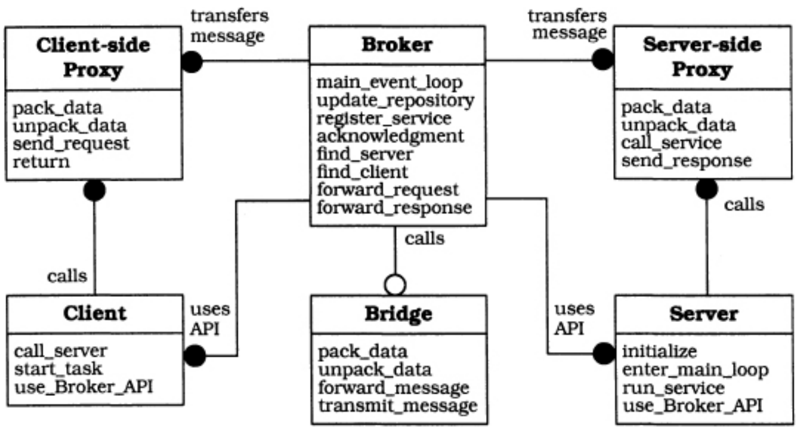


Class Bridge	**Collaborators**
Responsibility • Encapsulates network-specific functionality. • Mediates between the local broker and the bridge of a remote broker.	• Broker • Bridge

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:

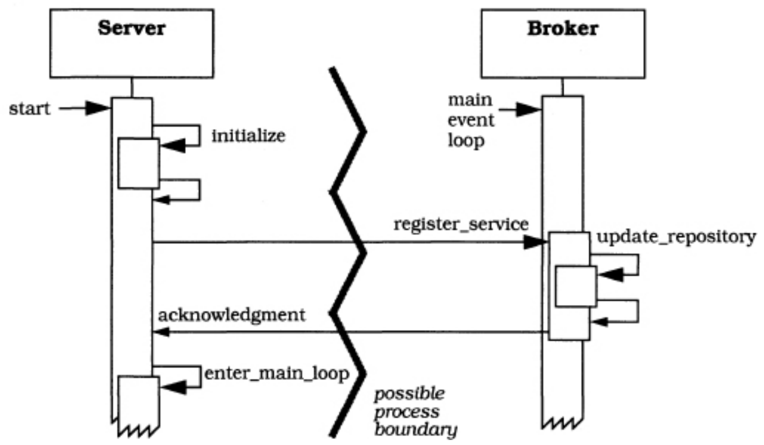


Dynamics

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

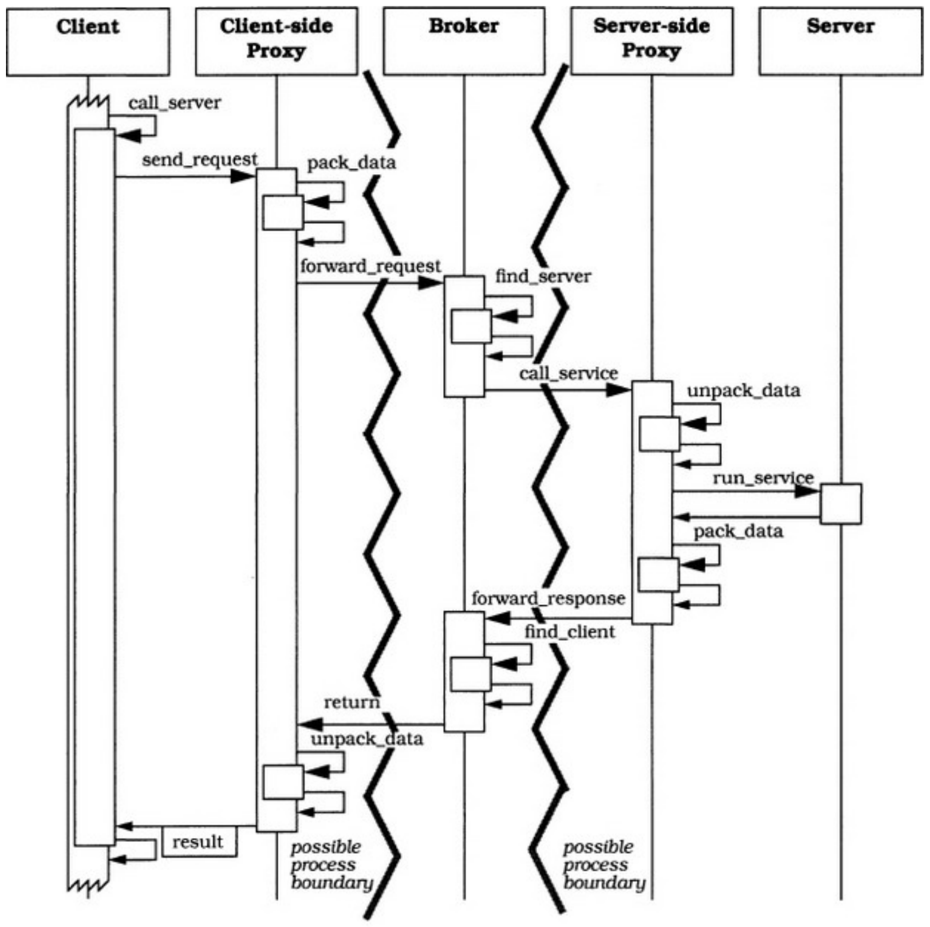
Scenario I illustrates the behavior when a server registers itself with the local broker component:

- 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.



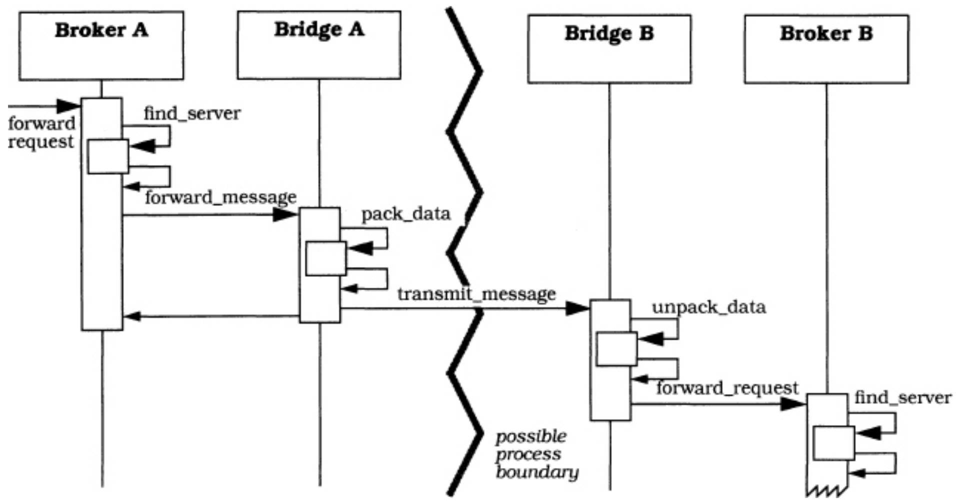
Scenario II illustrates the behavior when a client sends a request to a local server. In this scenario we describe a synchronous invocation, in which the client blocks until it gets a response from the server. The broker may also support asynchronous invocations, allowing clients to execute further tasks 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).

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:

5.1 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.
5.2 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.
5.3 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.
5.4 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.
5.5 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.
5.8 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 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 us 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 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.