Architectural patterns

Contents

Architectural patterns

INTRODUCTION

Before major development starts, we need to choose an architecture that will provide us with the desired quality attributes. Therefore we need a way to discuss architectural options and their quality consequences in advance, before they can be applied to a design. Our decisions at this stage can only be based on experience with architectural choices in previous systems. Such architectural experience, reduced to its essence and no longer cluttered with the details of the systems that produced it, is recorded in architectural patterns.

Many systems have a similar structure. Distributed systems for instance, can have a client-server structure, with clients making requests, and a server processing those requests and answering them. When we observe a similarity, we want to know what is common among the solutions, and what variations are possible. We ask in what circumstances an approach may be used, and how it should be used and customized for a particular system. An architectural pattern provides an answer to such questions.

This learning unit introduces architectural patterns and their relation with design patterns. We will discuss several examples of architectural patterns, but these examples do not exhaust all architectural patterns: we will discuss ways to classify them.

With an example, we will show the effect of choosing different patterns for the same problem. The example makes clear that you need to be able to choose the right pattern for a given situation. To make the right choice you need experience with different systems. Knowledge about patterns will help you to expand upon what you learn by experience, because by learning about patterns, you learn from the experience of others.

LEARNING GOALS
After having studied this learning unit you should be able to:
- describe the structure and function of the patterns which are covered in this unit,
- describe the advantages and disadvantages of the patterns which are covered in this unit,
- explain the difference between a style and a pattern,
- give examples of applications of the patterns covered in this unit,
- name examples of patterns fitting in a certain style.

KERNEL

3.1 Patterns

Definition 16 (Architectural pattern) *An architectural pattern is a proven structural organization schema for software systems.*

A pattern is a description of a set of predefined subsystems and their responsibilities. In a system structured according to the client-server pattern, for instance, two subsystems are distinguished: the client (which can have many instances), and the server (which is unique). The responsibility of the client may be to show a user-interface to the user; the responsibility of the server may be to process lots of questions, and to guard data that are of interest to the client.

A pattern also describes rules and guidelines for organizing the relationships among the subsystems. The relationship between client and server is that the client asks questions and the server answers them.

Patterns are written by people with lots of experience. Patterns make knowledge which could have remained hidden in the heads of these experienced people explicit. This enables others to learn from those experiences. Patterns are not constructed by a single person: they reflect the experience of many developers. They capture existing, well-proven solutions in software development, and help to promote good design practices.

Architectural patterns are also called *styles*, or standard architectures, but the word architectural style is more often used for a concept less fine-grained than a pattern; several patterns may then belong to the same architectural style. We will explain the subtle differences later in this learning unit.

3.1.1 Why are patterns helpful?

When a certain kind of problem is solved by many developers in a similar way, and it is generally accepted that this way solves that problem well, it becomes a pattern. So, a pattern addresses a recurring design problem, for which a general solution is known among experienced practitioners: a pattern documents existing, well-proved design solutions.

By writing a pattern, it becomes easier to reuse the solution. Patterns provide a common vocabulary and understanding of design solutions. Pattern names become part of a widespread design language. They remove the need to explain a solution to a particular problem with a lengthy description. Patterns are therefore a means for documenting software architectures. They help maintaining the original vision when the architecture is extended and modified, or when the code is modified (but can not guarantee that).

Patterns support the construction of software with defined properties. When we design a client-server application, for instance, the server should not be built in such a way that it initiates communication with its clients.

Many patterns explicitly address non-functional requirements for software systems. For example, the MVC (Model-View-Controller) pattern supports changeability of user interfaces. Patterns may thus be seen as building blocks for a more complicated design.

3.1.2 Pattern schema or template

Patterns are described using a *pattern template* or *schema*. All of the many different templates have at least the following components:

- *Context*: the situation giving rise to a problem.
- *Problem*: the recurring problem in that context. A solution to the problem should fulfill requirements, consider constraints, and have desirable properties. These conditions are called forces. Forces may conflict with each other (performance may conflict with extensibility, for instance). Forces differ in the degree in which they are negotiable.
- *Solution*: a proven solution for the problem. The solution is given as a structure with components and relationships, and as a description of the run-time behavior. The first description is a static model of the solution; the second is a dynamic one.

3.1.3 Design patterns and architectural patterns

What is the difference between *design patterns* and architectural patterns? Design patterns offer a common solution for a common problem in the form of classes working together. They are thus smaller in scale than architectural patterns, where the components are subsystems rather than classes.

Design patterns do not influence the fundamental structure of a software system. They only affect a single subsystem. Design patterns may help to implement an architectural pattern. For example, the observer pattern (a design pattern) is helpful when implementing a system according to the MVC architectural pattern.

The concept of patterns was originally introduced by Christopher Alexander in building architecture, in 'A pattern language' [3] (1977), and 'The timeless way of building' [2] (1979). Design patterns first attracted attention in software design and development (the *Gang of Four* book [18]). Since then, patterns have been used in more disciplines: there are analysis patterns, user interface patterns, programming idioms, functional design patterns, and so on.

3.2 Examples of architectural patterns

In this section, we describe several architectural patterns. For each we describe the components and connections involved, give one or more usage examples, and discuss advantages, disadvantages and other issues.

3.2.1 Layers pattern

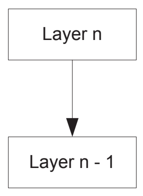

Figure 3.1: Layers of different abstraction levels

The *layers architectural pattern* (see Figure 3.1) helps to structure applications that can be decomposed into groups of subtasks, each of which is at a particular level of abstraction. Each layer provides services to the next higher layer. Services in a layer are implemented using services from the next lower layer. Service requests are frequently done by using synchronous procedure calls.

In short, this pattern means that conceptually different issues are implemented separately, and that layers of a higher abstraction level use services of a lower abstraction level, and not the other way around.

This pattern has the following benefits:

– A lower layer can be used by different higher layers. The TCP layer from TCP/IP connections for instance, can be reused without changes by various applications such as telnet or FTP.
– Layers make standardization easier: clearly defined and commonly accepted levels of abstraction enable the development of standardized tasks and interfaces.
– Dependencies are kept local. When a layer shows the agreed interface to the layer above, and expects the agreed interface of the layer below, changes can be made within the layer without affecting other layers. This means a developer can test particular layers independently of other layers, and can develop them independently as well: development by teams is supported this way.

A result of the above is that layers may easily be replaced by a different implementation.

Example: networking protocols

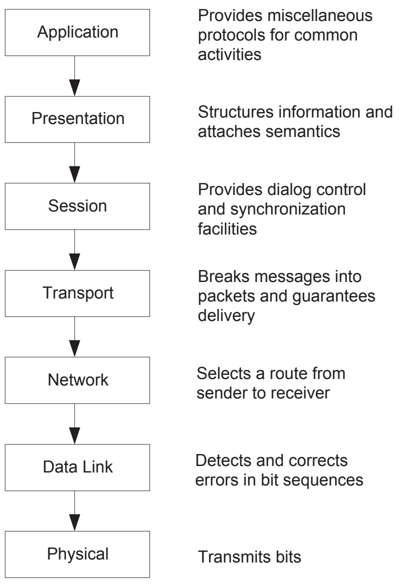
Application
Provides miscellaneous protocols for common activities
Presentation
Structures information and attaches semantics
Session
Provides dialog control and synchronization facilities
Transport
Breaks messages into packets and guarantees delivery
Network
Selects a route from sender to receiver
Data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

Figure 3.2: Example: networking protocols

Figure 3.2 depicts the ISO Open System Interconnect seven layer protocol (the *ISO/OSI protocol*). TCP/IP is a simplified version of this protocol. FTP and HTTP are examples from the application layer; TCP is the transport layer, and IP is the network layer.

Some other examples of this pattern are:

– In the Java virtual machine the application in Java consists of instructions for the Java virtual machine; the JVM uses services from the operating system underneath.
– The standard C library is built on Unix system calls.
– Web application systems often show four or more layers: presentation, application logic, domain logic, and data.

- The microkernel architecture has layers on top of the microkernel: The Mach operating system, the JBoss application server, Windows NT and QNX are examples of the microkernel architecture.

Issues in the Layers pattern

The most stable abstractions are in the lower layer: a change of the behavior of a layer has no effect on the layers below it. The opposite is true as well: a change of the behavior of a lower layer has an effect on the layers above it, so should be avoided.
Of course, changes of or additions to a layer without an effect on behavior will not affect the layers above it. Layer services can therefore be implemented in different ways (think of the bridge pattern here, where a dynamic link is maintained between abstraction and implementation).
Layers can be developed independently. However, defining an abstract service interface is not an easy job. There may also be performance overhead due to repeated transformations of data. Furthermore, the lower layers may perform unnecessary work, not required by the higher layer.
There are several variants of the Layers pattern.

- In a *Relaxed layered system*, each layer may use the services of all layers below it, not only of the next lower layer. This has efficiency benefits, but leads to a loss of maintainability. Windows' microkernel architecture is an example of this variant. Another example is formed by the user interface of Eclipse: Eclipse makes use of the SWT (the Standard Widget Toolkit, [14]) and JFace [13]. JFace is a layer on top of the SWT and offers a higher level of abstraction than SWT, but it is possible to use classes of both JFace and the SWT in the same application.
- Another variant is to allow *callbacks* for bottom-up communication: here, the upper layer registers a callback function with the lower layer, to be notified at the occurrence of an events.

3.2.2 Client-server pattern

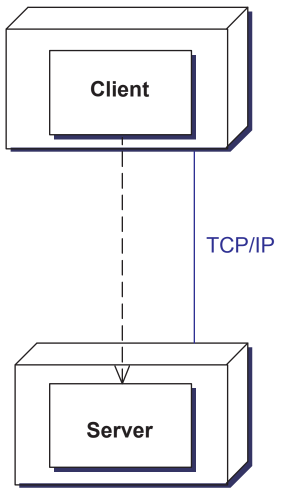

Figure 3.3: Client-server pattern

In the Client-server architectural pattern (see Figure 3.3), a server component provides services to multiple client components. A client component requests services from the server component. Servers are permanently active, listening for clients.

The requests are sent beyond process and machine boundaries. This means that some inter-process communication mechanism must be used: clients and servers may reside on different machines, and thus in different processes. In fact, you can see the Client-server pattern as a form of the layered pattern, crossing process or machine boundaries: clients form the higher level and the server forms the lower level.

Examples

Examples of the Client-server pattern are remote database access (client applications request services from a database server), remote file systems (client systems access files, provided by the server system; applications access local and remote files in a transparent manner) or web-based applications (browsers request data from a web server).

Issues in the Client-server pattern

Requests are typically handled in separate threads on the server.

Inter-process communication causes overhead. Requests and result data often have to be transformed or marshaled because they have a different representation in client and server, and there is network traffic.

Distributed systems with many servers with the same function should be transparent for clients: there should be no need for clients to differentiate between servers. When you type in the URL for Google for instance, you should not have to know the exact machine that is accessed (location transparency), the platform of the machine (platform transparency), or the route your request travels, and so on. Intermediate layers may be inserted for specific purposes: caching, security, load balancing for instance.

Sometimes, callbacks are needed for event notification. This can also be seen as a transition to the Peer-to-peer pattern.

State in the Client-server pattern

Clients and servers are often involved in 'sessions'. This can be done in two different ways:

- With a *stateless server*, the session state is managed by the client. This client state is sent with each request. In a web application, the session state may be stored as URL parameters, in hidden form fields, or by using cookies. This is mandatory for the REST architectural style [15] for web applications which is described in more detail further in this unit; see also the learning unit on REST.
- With a *stateful server*, the session state is maintained by the server, and is associated with a client-id.

State in the Client-server pattern influences transactions, fault handling and scalability. Transactions should be atomic, leave a consistent state, be isolated (not affected by other requests) and durable. These properties are hard to obtain in a distributed world.

Concerning fault handling, state maintained by the client means for instance that everything will be lost when the client fails. Client-maintained state poses security issues as well, because sensitive data must be sent to the server with each request. Scalability issues may arise when you handle the server state in-memory: with many clients using the server at the same time, many states have to be stored in memory at the same time as well.

REST architecture

REST stands for Representational State Transfer. A REST architecture is a client-server architecture, where clients are separated from servers by a uniform interface. Servers offer addressable resources. Communication is stateless. A server may be stateful, but

in that case, each server-state should be addressable (for instance, by a URL). A REST architecture is also a layered system: for a client, it is transparent whether it is directly connected to a server, or through one or more intermediaries.
Web applications with stateless communication follow the rules of this pattern.

3.2.3 Master-slave pattern

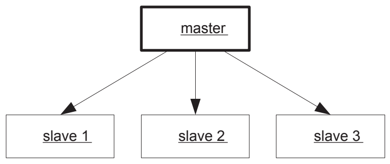

Figure 3.4: Master-slave pattern

The *Master-slave pattern* (see Figure 3.4) supports fault tolerance and parallel computation. The master component distributes the work among identical slave components, and computes a final result from the results the slaves return. Figure 3.5 shows a sequence diagram of a master distributing work between slaves.
The Master-slave pattern is applied for instance in process control, in embedded systems, in large-scale parallel computations, and in fault-tolerant systems.

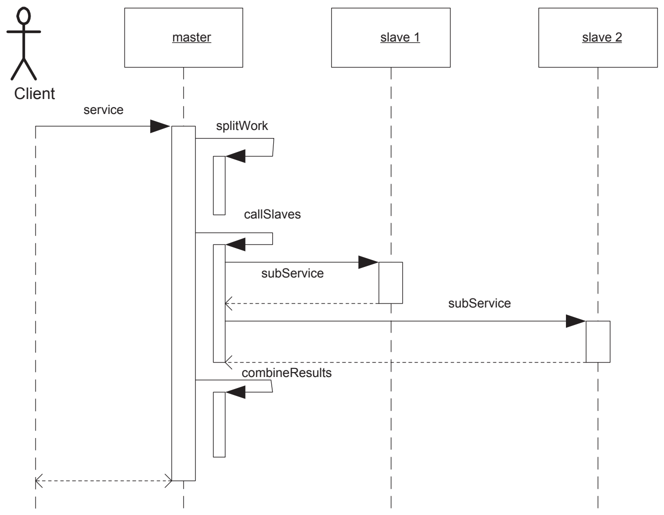

Figure 3.5: Sequence diagram for the master-slave pattern

Examples

An application area for the Master-slave pattern is fault tolerance: the master delegates the job to be done to several slaves, receives their results, and applies a strategy to decide which result to return to the client. One possible strategy is to choose the result from the first slave that terminates. Another strategy is to choose the result that the majority of slaves have computed. This is fail-proof with respect to slaves (the master can provide a valid result as long as not all slaves fail), but not with respect to the master. Failure of slaves may be detected by the master using time-outs. Failure of the master means the system as a whole fails.

Another application area is parallel computing: the master divides a complex task into a number of identical subtasks. An example is matrix computation: each row in the product matrix can be computed by a separate slave.

A third application area is that of computational accuracy. The execution of a service is delegated to different slaves, with at least three different implementations. The master waits for the results, and applies a strategy for choosing the best result (for instance the average, or the majority).

Issues in the Master-slave pattern

The Master-slave pattern is an example of the *divide-and-conquer* principle. In this pattern, the aspect of coordination is separated from the actual work: concerns are separated. The slaves are isolated: there is no shared state. They operate in parallel.

The latency in the master-slave communication can be an issue, for instance in real-time systems: master and slaves live in different processes.

The pattern can only be applied to a problem that is decomposable.

3.2.4 Pipe-filter pattern

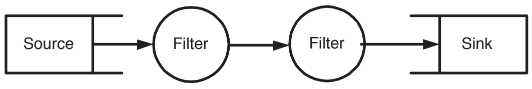

Figure 3.6: Pipe-filter pattern

The *Pipe-filter architectural pattern* (see Figure 3.6) provides a structure for systems that produce a stream of data. Each processing step is encapsulated in a filter component (a circle in Figure 3.6). Data is passed through pipes (the arrows between adjacent filters). The pipes may be used for buffering or for synchronization.

Examples

Examples of the Pipe-filter pattern are *Unix shell commands*, such as:

```
cat file | grep xyz | sort | uniq > out
```

This pattern divides the task of a system into several sequential processing steps. The steps are connected by the data flow through the system: the output of a step is the input for the next step. In the example, the `cat` filter reads the file and passes the contents of the file to the `grep` filter. The `grep` filter selects lines containing `xyz`, and passes these lines to the `sort` filter. The `sort` filter sorts the lines, and passes the sorted lines to the `uniq` filter. The `uniq` filter removes duplicate lines, and passes the result to `out`.

A filter consumes and delivers data incrementally (another name for the same concept is lazily): it produces output as soon as it comes available, and does not wait until all input is consumed.

Another example of the Pipe-filter pattern are compilers, where a lexical analyzer analyzes the source file, and sends the resulting tokens to a parser, which sends the resulting parse tree to a semantic analyzer, which produces an augmented syntax tree, that is used by the code generator to produce code, which may be optimized, and ultimately

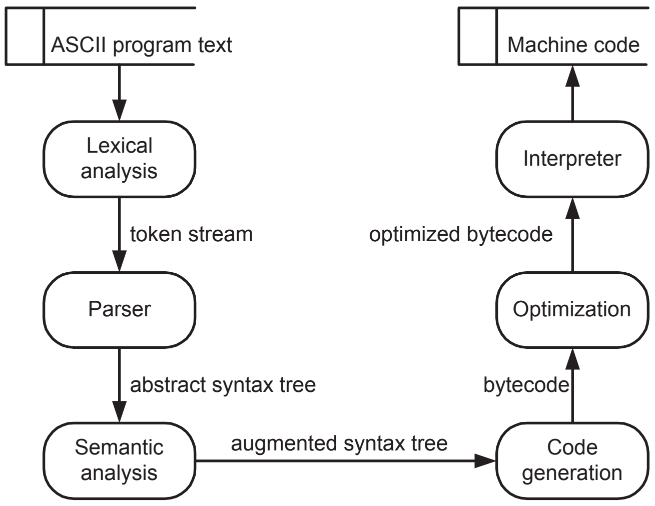

Figure 3.7: A compiler

translated into machine code (there may be more or fewer steps involved). Figure 3.7 shows the components of a compiler. In practice, compilers do not follow the pattern strictly: the steps share global data (the symbol table).

Issues in the Pipe-filter pattern

The Pipe-filter pattern has several nice properties, and some disadvantages. It is easy to add new filters: a system built according to the pipe-filter pattern is easy to extend. Filters are reusable: it is possible to build different pipelines by recombining a given set of filters. Because of the standard interface, filters can easily be developed separately, which is also an advantage. However, that same interface may be the cause of overhead because of data transformation: when the input and the output have the form of a string for instance, and filters are used to process real numbers, there is a lot of data-transformation overhead.

Filters do not need to store intermediate results in files, and need not share state. Input and output can come from, and go to different places. Another advantage of the Pipe-filter pattern is that it shows natural concurrent processing, when input and output consist of streams, and filters start computing when they receive data. Analysis of the behavior of a pipe-filter-based system is easy, because it is a simple composition of the behaviors of the filters involved. When the input is called `x`, the behavior of the first filter is described by function `g`, and the behavior of the second filter is described by function `f`, the result of the pipeline can be described as:

```
f(g(x))
```

Because of this composition property, it is possible to analyze throughput as well (throughput is determined by the slowest filter), and the possibility of deadlocks. Deadlocks may occur when at least one of the filters needs all data before producing output. In such a case, the size of the buffer may be too small, and the system may deadlock. The Unix `sort` filter is an example of such a filter.

A disadvantage of the Pipe-filter pattern, aside from the already mentioned possible data-transformation overhead, is that it is hard to use it for interactive applications.

3.2.5 Broker pattern

The *Broker pattern* is used to structure distributed systems with decoupled components, which interact by remote service invocations. Such systems are very inflexible when components have to know each others' location and other details (see Figure 3.8). A broker component is responsible for the coordination of communication among components: it forwards requests and transmits results and exceptions.

Servers publish their capabilities (services and characteristics) to a broker. Clients request a service from the broker, and the broker then redirects the client to a suitable service from its registry.

Using the Broker pattern means that no other component than the broker needs to focus on low-level inter-process-communication.

Interface Definition Language (IDL)

To give a textual description of the interfaces a server offers, an *Interface Definition Language* (IDL) is used. Examples of IDLs are OMG-IDL (Object Management Group, for CORBA), Open Service Interface Definitions, Microsoft .NET CIL (Common Intermediate Language) or WSDL (Web Service Description Language).

Alternate software may use a binary standard, like Microsoft OLE (Object Linking and Embedding) or Universal Network Objects for the OpenOffice suite. Such a binary description consists of a table with pointers to method implementations. It allows clients to call methods indirectly, using those pointers. A binary standard needs support from the programming language used.

Examples

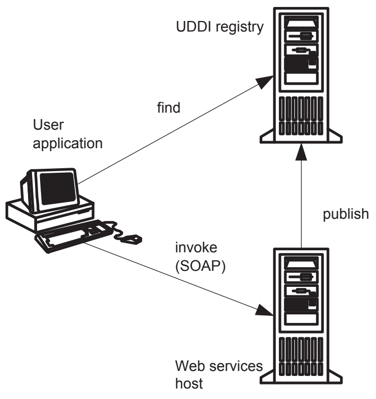

Figure 3.8: Web services

An example in which the Broker pattern is used is formed by *web services*. The Broker pattern in a web services application, as shown in Figure 3.8, is the server with the *UDDI registry*. UDDI stands for Universal Discovery, Description and Integration. It is a repository of web services. The IDL used for web services is WSDL. SOAP (Simple Object Access Protocol) is the transport protocol for messages, written in XML.

Another example of the Broker pattern is *CORBA*, for cooperation among heterogeneous object-oriented systems, and web services.

Issues in the Broker pattern

The Broker pattern allows dynamic change, addition, deletion and relocation of objects, and it makes distribution transparent to the developer. It requires standardization of service descriptions. When two brokers cooperate, bridges may be needed to hide

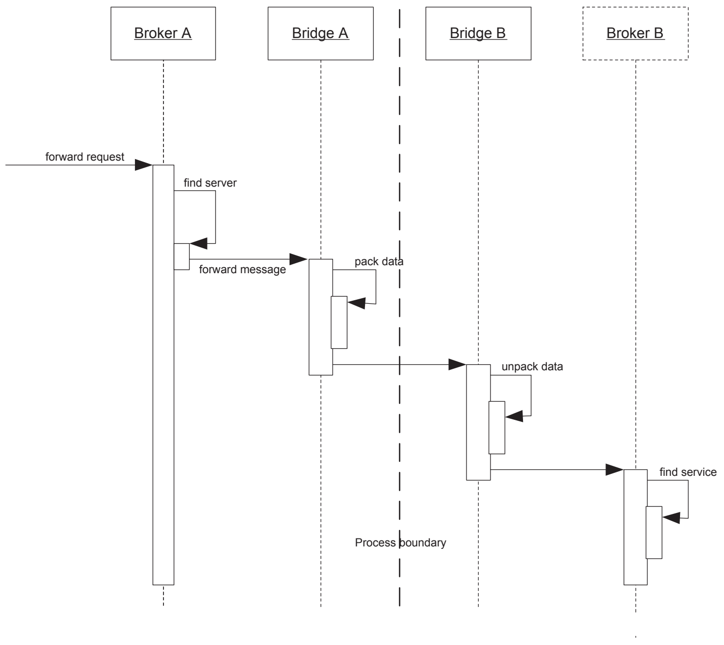

Figure 3.9: Two brokers cooperating

implementation details, as in Figure 3.9:

Broker A receives an incoming request for a certain service. 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 understandable 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 then performs all the actions necessary when a request arrives, as described in the first step of the scenario.

3.2.6 Peer-to-peer pattern

The *Peer-to-peer pattern* can be seen as a symmetric Client-server pattern: peers may function both as a client, requesting services from other peers, and as a server, providing services to other peers. A peer may act as a client or as a server or as both, and it may change its role dynamically.

Both clients and servers in the peer-to-peer pattern are typically multithreaded. The services may be implicit (for instance through the use of a connecting stream) instead of requested by invocation.

Peers acting as a server may inform peers acting as a client of certain events. Multiple clients may have to be informed, for instance using an event-bus.

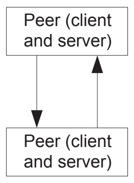

Figure 3.10: Peer-to-peer pattern

Examples

Examples of the Peer-to-peer pattern are the domain name system for internet, the distributed search engine Sciencenet, multi-user applications like a drawing board, or peer-to-peer file-sharing like Gnutella [50] or Bittorrent.

Issues in the Peer-to-peer pattern

An advantage of the peer-to-peer pattern is that nodes may use the capacity of the whole, whilee bringing in only their own capacity. In other words, there is a low cost of ownership, through sharing. Also, administrative overhead is low, because peer-to-peer networks are self-organizing.

The Peer-to-peer pattern is scalable, and resilient to failure of individual peers. Also, the configuration of a system may change dynamically: peers may come and go whilee the system is running.

A disadvantage may be that there is no guarantee about quality of service, as nodes cooperate voluntarily. For the same reason, security is difficult to guarantee. Performance grows when the number of participating nodes grows, which also means that it may be low when there are few nodes.

3.2.7 Event-bus pattern

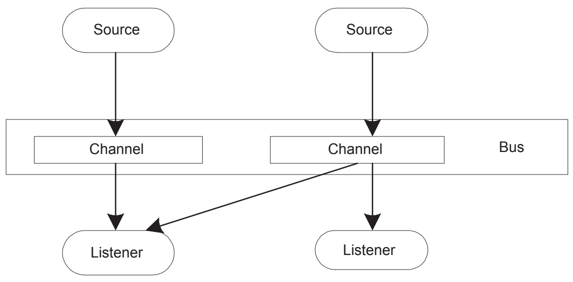

Figure 3.11: Event-bus pattern

The *Event-bus pattern* is a pattern that deals with events. It works as follows: event sources publish messages to particular channels on an event bus. Event listeners subscribe to particular *channels*. Listeners are notified of messages that are published to a channel to which they have subscribed.

Generation and notification of messages is asynchronous: an event source just generates a message and may go on doing something else; it does not wait until all event listeners have received the message.

Channels may be implicit, for instance using the event pattern, implemented in the Java event model. An explicit channel means that a subscriber subscribes directly with a specific named publisher; an implicit channel means that a subscriber subscribes to a specific named channel (to a particular event type in the Java event model), and does not need to know which producers produce for that channel.

Examples

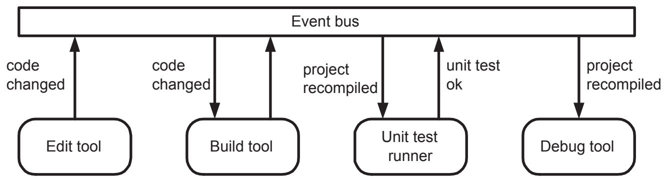

Figure 3.12: Software development environment

The Event-bus pattern is used in process monitoring, in trading systems, and in software development environments as is shown in Figure 3.12. Another example is real-time data distribution middleware, like OpenSplice [41].

Issues in the Event-bus pattern

The Event-bus pattern has the following characteristics. New publishers, subscribers and connections can be added easily, possibly dynamically. Delivery issues are important: for the developer of an event listener, it is important to realize that assumptions about ordering, distribution, and timeliness are hard to make. Also, scalability may be a problem, as all messages travel through the same event bus: with an increase of the number of messages, the capacity of the event bus may become the bottleneck.

The Event-bus pattern also allows several variations. The bus can provide event transformation services, for instance, or the bus can provide coordination, to script specific tasks.

3.2.8 Model-view-controller pattern

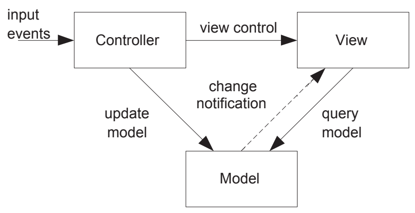

Figure 3.13: Model-view-controller

In the *Model-View-Controller pattern*, or MVC pattern (see Figure 3.13), an interactive application is divided into three parts: the model contains the core functionality and data, the view displays the information to the user (more than one view may be defined), and the controller handles the input from the user.

The MVC pattern is particularly suitable for multiple graphical user interfaces (GUIs). The model does not depend on the number and kind of GUIs, so the pattern allows for easy changes to the 'look and feel'.
Consistency between model and view is maintained through notification. The MVC pattern often uses the *observer design pattern*. User input can invoke a change in the model, and a subsequent change in what the view displays, as is shown in the sequence diagram of the MVC pattern in Figure 3.14.

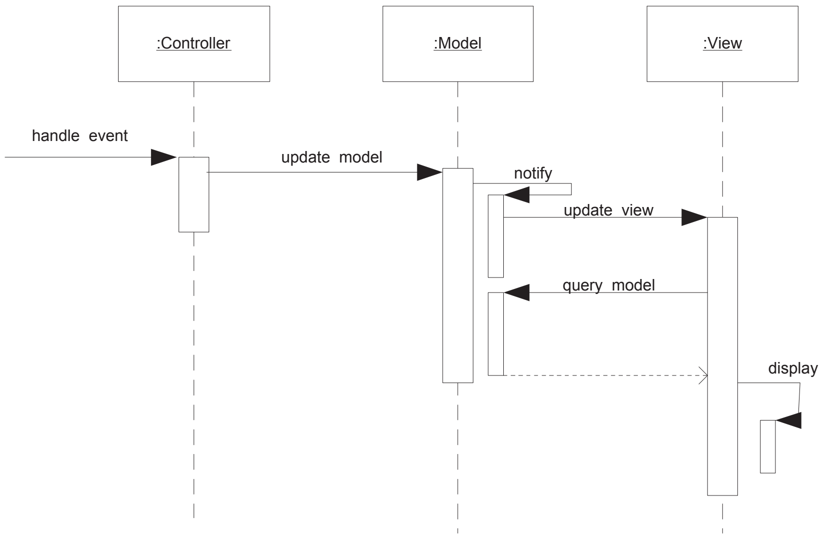

Figure 3.14: Sequence diagram of the MVC pattern

Examples

The MVC pattern was introduced with the Smalltalk programming language. It was called a paradigm then: the concept of patterns did not exist.
Examples of the MVC pattern are web presentation (see the learning unit on patterns for enterprise application architecture), and the document-view architecture of Windows applications, which enables users to see for instance Word or PowerPoint documents in different views (think of print layout, web layout, overview).

Issues in the MVC pattern

The MVC pattern makes it easy to have multiple views of the same model, which can be connected and disconnected at run-time. It is possible to base an application framework on this pattern. Smalltalk development environments already did this.
However, the MVC pattern increases complexity. Not all visible elements lend themselves for separation of model, views and control: menus and simple text elements may be better off without the pattern. Also, the pattern potentially leads to many unnecessary updates, when one user action results in different updates.
View and control are separated, but are very closely related. In practice, they are often put together. Views and controllers are also closely coupled to the model. In web applications, a change in the model (for instance adding an e-mail address to data about persons) will lead to a change in the view and controller as well (the web site will have to show the property, and the possibility to change the property should be added as well).

3.2.9 Blackboard pattern

The *Blackboard pattern* is useful for problems for which no deterministic solution strategies are known. Several specialized subsystems assemble their knowledge to build a possibly partial or approximate solution.

All components have access to a shared data store, the blackboard. Components may produce new data objects that are added to the blackboard. Components look for particular kinds of data on the blackboard, and may find these by pattern matching.

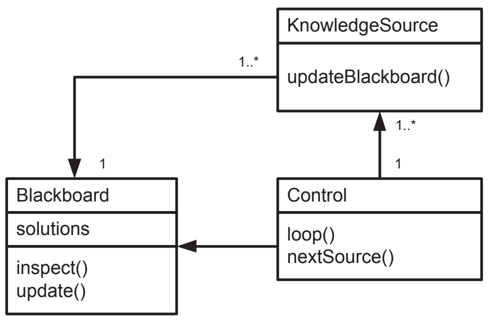

Figure 3.15: Blackboard pattern

Examples

Examples of problems in which the Blackboard pattern can be usefully applied are speech recognition, submarine detection, or inference of the 3D structure of a molecule. Tuple Space systems, like JavaSpaces, form another example of this pattern.

Issues in the blackboard pattern

Adding new applications is easy. Extending the structure of the data space is easy as well, but modifying the structure of the data space is hard, as all applications are affected. Furthermore, processes have to agree on the structure of the shared data space. There may be a need for synchronization and access control.

3.2.10 Interpreter pattern

The *Interpreter pattern* is used for designing a component that interprets programs written in a dedicated language. The interpreted program can be replaced easily.

Examples

Examples of the interpreter pattern are rule-based systems like expert systems, web scripting languages like JavaScript (client-side) or PHP (server-side), and Postscript.

Issues in the Interpreter pattern

Because an interpreted language generally is slower than a compiled one, performance may be an issue. Furthermore, the ease with which an interpreted program may be replaced may cause a lack of testing: stability and security may be at risk as well.

On the other hand, the interpreter pattern enhances flexibility, because replacing an interpreted program is indeed easy.

3.3 Applying patterns: KWIC example

A classical example to illustrate the differences between architectural patterns is the KWIC problem: *KeyWord In Context*, introduced by Parnas [39]. In this section we introduce this problem, and show how different architectural patterns can be used to solve it [47].
The keyword in context problem takes as input a list of lines; a line is composed of words. The output of the solution for the problem should be all 'circular shifts' of all lines, sorted. A circular shift is done by repeatedly removing the last word and appending it at the beginning of the line. For example, with the input:

```
man eats dog
```

the output should be:

```
dog man eats
eats dog man
man eats dog
```

3.3.1 Shared data

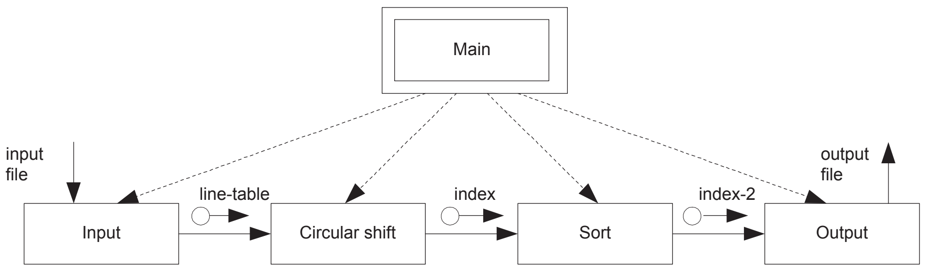

Figure 3.16: KWIC, the classical solution

Traditionally, the KWIC problem is solved using shared data and functional decomposition. Figure 3.16 is a *Yourdon structure chart* [55] for the classical solution to the KWIC problem. The processing elements are subroutines, and an arrow means an invocation. The small arrows show which data is added to the shared data.
The Main subroutine invokes functions. The first invoked function reads the input file, and adds a table of lines to the shared data. The second function performs the circular shift, and adds the various shifted lines to the shared data (as a series of indexes to indicate the order). The third function then sorts all these lines, and adds the resulting lines to the shared data (once again using indexes). The last invoked function writes the result to an output file.

3.3.2 Layers pattern

An alternate solution to the KWIC problem uses the Layers pattern. In Figure 3.17, the elements are objects, and the arrows denote method calls. No data is shared: communication of data is done through method calls only.
The advantage of this solution is that the data is hidden inside the objects, which means that choices for data structures and algorithms are encapsulated in specific components, and may therefore be more easily changed, by keeping the abstract interfaces as they are.

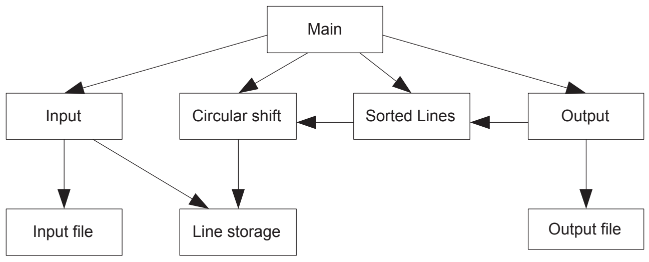

Figure 3.17: KWIC, Layers pattern

The organization in layers is based on objects only calling methods from objects in the same layer or in the layer directly beneath them.

3.3.3 Event-bus

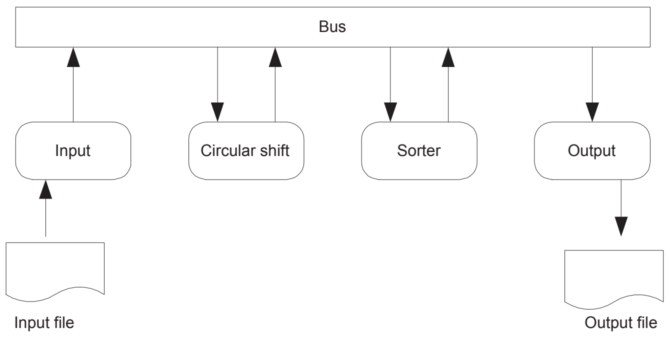

Figure 3.18: KWIC, Event bus pattern

Another solution to the KWIC problem uses the Event-bus architecture. Components that are interested in certain events are notified when those events become available. That means that components are called implicitly.
Figure 3.18 is a dataflow diagram [56]. Elements are processes, and the arrows are data in transfer between processes. The shifter will be invoked after the input process has inserted the first line. Data should be shared between the input process and the circular shift, and among the circular shift, the sorter and the output process. The sorter is invoked each time the circular shifter has inserted a line in their shared data.
This solution does not offer the advantages of data hiding of the previous solution, but it can be quicker because there may be parallelism. However, it is a complicated solution, with two shared data spaces and an event bus.

3.3.4 Pipe-filter

Yet another possible solution to the KWIC problem uses the Pipe-filter pattern. In this case a chain of transformers is constructed. We need a uniform data format to do that.
The same amount of parallelism as has been achieved in the event bus solution is possible here, in a less complicated system.
The same amount of data hiding as in the layered object-oriented solution is possible, with the only drawback that there is a uniform data format for the pipes. This makes

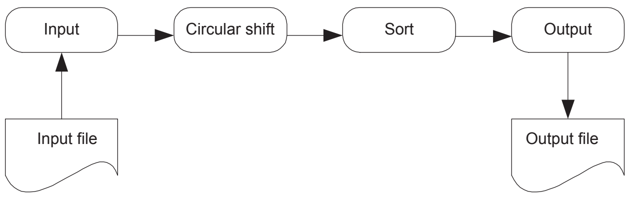

Figure 3.19: KWIC, Pipe-filter pattern

the filters easier to work with, but when different data structures are chosen inside the filters, there may be overhead because of translation between different data structures.

3.4 Architectural styles

Patterns have been developed bottom-up: for a given problem, a certain kind of solution has been used over and over again, and this solution has been written down in the form of a pattern.

Architectural styles on the other hand, have been formulated top-down: when you see a software system as a configuration of components and connectors, you can classify them according to the nature of the components and connectors. In general, patterns will belong to one of those styles.

Mary Shaw and Paul Clements have proposed a classification of styles [46], based on the constituent parts (components and connectors), control issues and data issues. This classification is as follows:

- *Interacting processes* have their own thread of control. Communication may have the form of asynchronous message passing, implicit invocation through events, or remote procedure calls. When the Event bus pattern is implemented with independent processes or with objects with their own thread of control, this pattern is an example of this style. The Client-server pattern and Peer-to-peer pattern are examples as well.
- In the *Dataflow style*, data flows from one component to another, in a stream. A pattern that belongs to this style is the Pipe-filter pattern. Some instances of the Client-server pattern also belong to this style, for instance when a client is used to receive and display streaming audio or video sent by a server.
- The *Data centered style* means that data is stored centrally. An example of this style is the Blackboard pattern. Again, instances of the Client-server pattern may belong to this style, when the main function of the server is to manage a database, and clients are used to access that database.
- In the *Hierarchical style*, the system is partitioned into subsystems with limited interaction. Patterns within this style are the Interpreter and the Layers pattern.
- The *Call and return style* has the calling process wait for the return of request. Patterns within this style are the Master-slave and, again, the Layers pattern. Object-oriented systems without threads are also an example of this style.

3.4.1 Choosing a style or a pattern

Which architectural pattern is best for a given situation depends on which requirements have the highest priority, such as:

- *Maintainability:* How easy or difficult is it to add an additional processing component, for instance to filter certain words? How easy or difficult is it to change the

input format, such as adding line numbers? In the Pipe-filter pattern for instance, adding a filter is very easy, but changing the input format might be hard.
- *Reusability:* Can individual components be reused in other systems? In this case, the Pipe-and-filter pattern enhances reusability because of the uniform data format that is used.
- *Performance:* Is the response time small enough? Is overall resource behavior (memory usage in this example) acceptable? Patterns that make use of parallelism, such as the Pipe-filer pattern and the Event-bus pattern, will have better performance. On the other hand, starting a complex system like the event bus system, or transforming data in every filter using a different data structure, may lower performance.
- *Explicitness:* Is it possible to provide feedback to the user? Per stage? This is not possible in the Pipe-filter pattern for instance.
- *Fault tolerance:* For the KWIC example, there is no difference between the different solutions, but had a Master-slave pattern been applied, fault-tolerance would have been enhanced.

The list of requirements and their priorities will vary for every system. No rigid guidelines can be given to tell you which pattern will be the best in every case. Much also depends on the implementation of the pattern. Independent processes, for example, may be implemented using threads or using processes on different machines. The balance between communication and computation, the capacity of the processors involved and the speed of communication between machines, among others, will decide which implementation will have the best performance.

DISCUSSION QUESTIONS

1. In a previous learning unit, we introduced the name tactic for an architectural decision that influences the quality of the product. Architectural patterns can be viewed as being constructed from tactics. Can you give some examples?
2. Once a pattern has been used in an architectural design, the reason for its use may be forgotten and over time the design may start to diverge from the pattern. Do you think this should be prevented? If so, how?
3. Design patterns often represent crosscutting concerns that can be implemented using aspect-oriented programming or program transformation techniques. Is the same true of architectural patterns?

SELF TEST

1. The workbook mentions that the REST architecture is a layered system. Explain why this is the case.
2. Describe a difference between the client-server pattern and the master-slave pattern with respect to the control flow.
3. In Figure 1.1 of the article about the Google architecture, one can recognize the Master-slave pattern. Which advantages of that pattern will probably have played a role in the decision to use it here?

FEEDBACK

1. The only explanation in the workbook is that it is transparent for the client whether it is connected directly with a server or through a series of intermediate servers. That means that the client may be viewed as a layer using the layer below, through a fixed interface. How that layer below is implemented (a direct connection to a server, or a series of intermediate servers) is irrelevant for the layer of the client.
2. In the client-server pattern the control flow is from the client to the server: the client asks a question, takes the initiative. In the master-slave pattern, the control flow is from the master to the slaves: the master decides which has to be performed by which slave. Thee master takes the initiative. Several clients ask questions to one server, while one master controls several slaves.
3. Fault-tolerance (geographically diffused clusters, with duplicated data), performance (choose the cluster close by the user that is not overloaded), parallelism (by using this pattern, many queries may be performed simultaneously).