CHAPTER 12

System Architecture

Learning Objectives

In this chapter you will learn

- what is meant by architecture in information systems development
- the factors that influence the architecture of a system
- the range of architectural styles that can be used as the basis of an architecture.

12.1 Introduction

In Chapter 5 we described the Unified Software Development Process (USDP) (Jacobson et al., 1999) as 'architecture-centric' and at several points in previous chapters we have mentioned the idea of producing architectural models of the proposed information system. In Chapter 8 we introduced the idea of patterns and the use of architectural patterns such as the Model–View–Controller architecture that is embodied in the use of Boundary, Control and Entity classes that we have used in the class models that we have produced. However, up to this point we have not defined what we mean by architecture in the context of information systems; nor have we explained how to design the architecture of a system.

Systems architecture is a broad topic and one that is the subject of many books. In this chapter we aim to give an introduction to systems architecture and to explain why it is important. Every system has an architecture of some sort. If the designers and developers do not take the time or have the skills to produce explicit architectural models of the system, the system will still have an architecture. However, that architecture will be implicit and will be influenced by factors such as the choice of programming language, database and platform, and the skills and experience of the development team. Any such implicit architecture is likely to result in a system that does not meet the non-functional requirements and is difficult to maintain or enhance. Producing an explicit architecture means that

the architect has to consider the non-functional requirements, the context of the system and how it and its components may be used and further developed in the future.

In the rest of this chapter, we explain what is meant by systems architecture, what are the factors that influence the development of an architecture and the kind of issues that are addressed by an architecture.

12.2 What Do We Mean by Architecture?

The use of the term architecture in the development of information systems obviously derives from the practice of architecture in the built environment. The Royal Institute of British Architects (RIBA) describes 'What Architects Do' as follows.

> Architects are trained to take your brief and can see the big picture–they look beyond your immediate requirements to design flexible buildings that will adapt with the changing needs of your business.
>
> Architects solve problems creatively–when they are involved at the earliest planning stage, they gain more opportunities to understand your business, develop creative solutions, and propose ways to reduce costs.

If we replaced the word 'buildings' with 'information systems' many systems architects and software architects would happily sign up to this definition of what they do. There are certain key features in these two sentences that apply as much to systems architecture as to the architecture of buildings.

- Systems architects act on behalf of the client. Part of their role is to understand the client's business and how that business can best be supported by an information system. However, the client may make conflicting demands on the new information system, and part of the systems architect's role is to resolve those conflicts.
- Systems architecture addresses the big picture. The architecture of an information system is a high-level view of the system: it is modelled in terms of the major components and the way they are interconnected; it does not normally address the detailed design of the system, though it may set standards to be applied.
- If flexibility is important, then systems architects will produce an architecture that is intended to deliver this quality. In the current climate of rapid change in the business environment, flexibility is often cited as a reason for adopting certain types of systems architecture. However, there are other qualities of information systems that may be more important for a particular client, in which case those qualities will be addressed by the architecture.
- Systems architects are concerned with solving problems. In information systems development, problems manifest themselves in terms of risks to the success of the project. The reason that the Unified Process is architecture-centric is that by concentrating on the architecture and making architectural decisions early in the project lifecycle, the risks can be reduced or mitigated.
- Reducing costs is not a primary objective of systems architects. However, proposing unnecessarily expensive solutions never wins anyone any friends, and

producing an explicit architecture for a new system means that the specific needs of that system are addressed and unnecessary features eliminated. It also means that risks are tackled early in the project lifecycle and that the chance is minimized of discovering late in the project that the new system will not meet some requirement, with the need for costly design changes or reworking.

Of these, probably the most important is that architecture is about the big picture. Analysis is inevitably about detail: the business analyst needs to understand and document every requirement in a clear and unambiguous way; the systems analyst must consider use cases and other requirements and translate them into a complete model of the classes necessary to support those use cases, their attributes and responsibilities or operations and a first-cut view of how instances of those classes will interact. Design is about translating every aspect of the analysis model into a design model that will effectively implement the requirements: the designer must consider the type of every attribute and design each operation to take the necessary parameters, return the right value and be efficient in its working. Architecture, on the other hand, looks at the large-scale features of the system and how those features work together as a whole: the architect groups classes together into packages, models the system as a set of interacting components and considers what platforms to deploy those components on in order to deliver the required qualities of the system.

There are a number of different views of architecture in the development of information systems. Our focus here is on systems architecture and software architecture. In Section 12.4 we discuss enterprise architecture and technical architecture and their relationship with systems and software architectures.

In their book on large-scale software architecture, Garland and Anthony (2003) use the definition of architecture from the Institute of Electrical and Electronics Engineers (IEEE) standard IEEE 1471–2000 (IEEE, 2000). This provides the following definitions of key terms.

- **System** is a set of components that accomplishes a specific function or set of functions.
- **Architecture** is the fundamental organization of a system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution[1].
- **Architectural Description** is a set of products that document the architecture.
- **Architectural View** is a representation of a particular system or part of a system from a particular perspective.
- **Architectural Viewpoint** is a template that describes how to create and use an architectural view. A viewpoint includes a name, stakeholders, concerns addressed by the viewpoint, and the modelling and analytic conventions.

Given this definition of architecture, then **Software Architecture** is the organization of a system in terms of its software components, including subsystems and the relationships and interactions among them, and the principles that guide the design of that software system.

1 From IEEE *Standard 1471–2000*, Copyright 2000 IEEE.

The IEEE definition is important because it stresses the fact that the same system can be shown in different views that emphasize different aspects of that system. Bass et al., (2003) point out that architecture is often defined as something like 'the overall structure of the system', but criticise this because it implies that a system has only a single structure. They suggest asking anyone who takes this position exactly which structure of the system the architecture represents.

Soni et al. (1995) identify four different aspects of software architecture, which are shown in Fig. 12.1.

In terms of object-oriented development, the conceptual architecture is concerned with the structure of the static class model and the connections between the components of the model. The module architecture describes the way the system is divided into subsystems or modules and how they communicate by exporting and importing data. The code architecture defines how the program code is organized into files and directories and grouped into libraries. The execution architecture focuses on the dynamic aspects of the system and the communication between components as tasks and operations execute.

The Rational Unified Process uses five views of the system, known as the '4 + 1 views' (Kruchten, 2004). The four views are the *logical view*, the *implementation view*, the *process view* and the *deployment view*. The one view that ties them all together is the *use case view*. These five views are explained in Fig. 12.2.

These five views conform to the IEEE 1471 definition of what constitutes a view. They provide a description of the system from a particular perspective. The static structural relationships between classes and packages in the logical view present a different aspect of the system from the dynamic relationships between runtime processes in the process view. A single diagram or model cannot easily combine both these perspectives.

Different views are like different maps of a country. It is possible to find maps that show the physical topography—mountains, hills, rivers and lakes; maps that show the human geography—towns, cities and the road and rail networks; maps that show the land use—farming, woodland, industry and human settlements; and maps that show schematically the volume of transport flow between major conurbations. However, trying to combine all these views of the country in a single map would make it confusing and difficult to understand.

Maps conform to particular conventions for how they represent the geography of a country. For example, the physical topography is shown using contour lines, colour or shading, or some combination of these three, to represent the height of features and the location of water. Clearly, models that represent different views of

Type of architecture	Examples of elements	Examples of relationships
Conceptual	Components	Connectors
Module	Subsystems, modules	Exports, imports
Code	Files, directories, libraries	Includes, contains
Execution	Tasks, threads, object interactions	Uses, calls

Figure 12.1 Four aspects of software architecture according to Soni et al. (adapted from Weir and Daniels, 1998)

View	Explanation
Use case view	The important use cases in the system and scenarios that describe architecturally significant behaviour
Logical view	Important design classes and interfaces in a package structure, with composite structure diagrams
Implementation view	Architectural decisions made for the implementation in terms of subsystems and components and relationships among them
Process view	A description of the processes (operating system processes and threads) and inter-process communications using stereotyped classes
Deployment view	Physical nodes for the likely deployment platform, components deployed on the nodes and the communication channels between them, using deployment diagrams

Figure 12.2 The 4 + 1 views.

a system must adopt some conventions for the different features that are shown in the model. The use of conventions makes it possible for the systems architect to communicate with stakeholders about the system and to provide guidance to designers and developers. A set of conventions for drawing architectural models is known as an *architecture description language* (ADL). Bass et al. (2003) present their own ADL, which consists of elements of four types, representing features in systems architectures, software architectures, process models and reference models. However, we would recommend using UML as an ADL. UML 2.0 has specific features that have been added and adapted in order to make it more suitable for modelling architectures as well as producing analysis and design models. The UML 2.0 Request for Proposals (OMG, 2000), which solicited proposals for the changes that should be made to the specification, had as one of its specific objectives the following.

> Enable the modeling of structural patterns, such as component-based development and the specification of run-time architectures.

This has resulted in the introduction of composite structure diagrams and changes to the component diagram notation.

12.3 Why Produce Architectural Models?

A software architect uses architectural models based on different views in order to reason about the proposed system and the way it will operate from different perspectives. In particular, this makes it possible to assess how well the system will deliver the non-functional requirements. Bass et al. (2003) do not like the term non-functional requirements. They argue that what they term *quality attributes* of a system, such as performance, security or fault tolerance, are intimately bound up with the behaviour of the system and the way that it responds to inputs. They believe that defining a set of non-functional requirements that are somehow separate from the functional behaviour of the system is dangerous, as it implies that the functionality of the system can be addressed first and then the non-functional

requirements can be tacked onto the system towards the end of the development process. We have used the term non-functional requirements because it is widely understood and because it focuses attention during requirements gathering on all those aspects of how well the system will deliver the functionality. However, we do not believe that this is a licence to ignore such requirements until the end of the development process.

Development processes based on the Unified Process are architecture-centric. This means that getting the architecture right is a priority, and this in turn means that from the start of the project the architects are trying to address the non-functional requirements of the system, because the architecture provides the framework for delivering these quality attributes of the system. Getting the architecture right early on is also about reducing the risks in the project. If one of the requirements of a new system is that it should be able to handle very large peak processing volumes (for example, in an online order processing system), then it is important to prove as soon as possible in the project that the architecture supports the achievement of these peak loads. If the early work addresses only the ability to process orders but does not ensure that the design can be scaled up to handle the peak loads, then there is always the risk that the fact that the system cannot handle the loads will not be discovered until late in the project, and that this will result in delays while the software is redesigned to cope with the peak volumes.

Using architectural models, the architect can assess the ability of the system architecture to deliver quality attributes such as high performance. The way that the different views in the 4 + 1 view of the system can contribute to assessing performance is shown in Fig. 12.3.

View	Contribution to assessing performance
Use case view	The use cases that require high performance can be identified and the scenarios used to walkthrough how the other views will affect the performance requirement.
Logical view	The logical view of classes will show whether techniques such as creating lightweight objects or value objects have been used to reduce the overheads associated with passing values around.
Implementation view	The more components or subsystems involved, the more likely there are to be communication overheads, so the implementation view should show a small number of components used in the process.
Process view	The process view can be used to assess how many running processes will exist, and whether there will be multiple instances of the same process so that the work can be shared out by a special process that handles load-balancing. The kind of interprocess communication that is used will affect how efficiently data can be passed between processes.
Deployment view	The deployment view will show where different components are deployed, and whether data has to travel from machine to machine, or whether all the processes needed to deliver a high-performance use case are located on the same machine.

Figure 12.3 The contribution of the 4 + 1 views to assessing performance.

It is important to realize that some of the features shown in Fig. 12.3 to increase performance will not contribute to the achievement of other quality attributes. For example, adding lightweight versions of classes will mean that for every business class there are two versions, and any change to the attributes of the business class means an associated change to the attributes of the lightweight version; this makes the code more complex to maintain. Similarly, reducing the number of components involved in a process may mean that functionality that does not naturally belong together is grouped into the same component or subsystem, and this reduces the flexibility of the system.

12.4 Influences on System Architecture

The systems architect developing the architecture for a new system does not operate in isolation. In any organization there will be existing systems that will constrain and influence the architecture. Many large organizations are now developing or have developed an *enterprise architecture*, which provides a framework for all system development. An enterprise architecture links the design of the business to the information systems that are needed to support that business. Either as part of an enterprise architecture or as a separate framework, many organizations have technology standards or a *technical reference architecture* that lays down a set of technologies, often including specific products that are acceptable, and defines how they should be used.

In the following subsections, we explain each of these influences in turn and the effect that they have on the architecture. In Section 12.5 we explain the range of *architectural styles* that are typically applied within the organization's information systems and that the architect can choose to adopt in developing new systems.

12.4.1 Existing systems

In many cases, the architecture of a new system will be designed to conform to the existing systems in the organization. This applies to the technical aspects such as choice of operating system, database and programming language, and to the way in which the components of the new system will be chosen, designed and interconnected. An organization that has adopted Java 2 Enterprise Edition (J2EE) or Microsoft .NET for its systems will expect new systems to be developed to fit in with this framework. Frameworks such as J2EE and .NET are well documented in books and web resources, but any business that adopts them is also likely to maintain a set of technology standards or a technical reference architecture that explains how to use the framework in the particular company.

Where there are existing systems, any new system may be able to take advantage of reuse of components in those systems. This is particularly the case when the new system and the old share the same architecture. In Chapter 8 we introduced the idea of reusable components, and we develop it further in Chapter 20. Organizations that plan for software reuse will typically use some kind of searchable repository in which they store reusable assets. The OMG, the body that manages the UML standard, also maintains the standard for the *Reusable Asset Specification* (RAS), which provides a set of guidelines about the structure, content and description of reusable software assets. Products such as LogicLibrary's Logidex

and Select Component Manager from Select Business Solutions provide tools to help manage collections of components.

Heritage systems

Sometimes the existing systems may not provide a pattern for development of new systems. The technologies that were used to develop them may be out of date and difficult to support. The systems may still be doing a good job of supporting the business, but a decision has been made to adopt new technologies. The term *heritage system* is sometimes used in preference to legacy system to describe a system that uses out-of-date technology but is still delivering a clear benefit to the business or is key to its operations. If a heritage system is not being replaced, the new system may need to access data from it via some kind of interface. *Enterprise Application Integration* (EAI) tools are software tools that connect to systems in order to integrate them with one another. If the heritage system uses a well-known technology, there is likely to be an *adapter* available that will connect it to the EAI tool and enable the EAI tool to extract data from the old system and make it available to the new or pass data into it in order to use its functionality.

Services

A technique for connecting to heritage systems that is growing in popularity is to wrap them in a layer of software that exposes the required functionality as *services*. *Web services* are the most recent technique applied to this problem, but the idea of a *Service-Oriented Architecture* (SOA) has been around for longer than the Web.

The wrapper acts as a service proxy, so that it looks the same as other services to the client systems that invoke operations on the service, as shown in Fig. 12.4.

There may be many kinds of interface to legacy systems. Sometimes they have been written to provide an Application Programming Interface (API), in which case it may be possible to use this, although there may be a limited choice of programming languages to use. Often they do not have an API, but may provide some other kind of interface: they listen on a TCP/IP socket for connections, or they check for files placed in a certain directory and treat the file as input. Sometimes the only way to access a legacy system is for the wrapper to pretend to be a terminal (*terminal emulation*) and connect to it and send text and terminal codes as though a user were typing the data in, and then to read the data that comes back and extract what is required from the mix of prompts, actual data values and control sequences. This is known as *screen-scraping*.

Reverse-engineering and model-driven architecture

Model-Driven Architecture (MDA) is one of the reasons for some of the changes that were made to UML to produce Version 2.0. The idea of MDA is to separate the business and application logic of a system from the underlying platform technology. This abstract view of what the system must do is known as a platform-

Figure 12.4 Wrapping a legacy system as a service.

independent model (PIM). The PIM is then combined with a definition of the platform architecture in order to produce a platform-specific model (PSM) that can be built and executed on a particular platform. In order to produce a PIM, it is necessary to be able to specify actions that must be carried out within a system in a precise and verifiable way. Using this approach, it should be possible to build a platform-independent specification of a system and then, using different standard mappings, to transform it into a platform-specific model. The PSM is then further transformed to implementation code using automated tools that are already available for building software from models. A single PIM could thus be implemented in different ways: J2EE, .NET, CORBA.

The OMG also has an initiative to promote MDA. UML is central to the MDA initiative. Although earlier versions of UML provided ways of modelling structures (class diagrams), interaction (sequence diagrams) and lifecycles (state machine diagrams), the specification of actions in activity diagrams was not up to the task of precisely defining how classes should carry out operations. The developers of UML 2.0 have added a precise *action semantics* to the language. Combined with the notation of activity diagrams (which have also been defined more precisely in UML 2.0), this is intended to make UML the language of choice for producing PIMs.

As well as creating applications by transforming PIMs, the OMG also promotes the idea of reverse-engineering existing applications into PIMs. The idea is that if the business and application logic of a legacy system can be separated from the implementation details, and represented in an abstract specification language (UML with action semantics), then that PIM can then be used to re-implement the functionality of the system on a different, more modern platform. Products such as ArcStyler from InteractiveObjects not only provide a way of producing implementations from PIMs, but also of reverse-engineering existing application code into a PIM.

12.4.2 Enterprise architectures

In large, complex organizations, particularly those that operate in many countries and have different divisions of the business that address different markets, there is a risk that system development will be unco-ordinated. Indeed, there is a risk that nobody will have an overall understanding of the business, let alone the systems that support it. When a project for a new system is proposed, it is difficult to analyse the effect of that new system. Questions that might be asked include the following.

- How does the system overlap with other systems in the organization?
- How will the system need to interface with other systems?
- Will the system help the organization to achieve its goals?
- Is the cost of the system justified?

This last question about the cost of the system is an important factor for any organization, but is particularly significant when the system is being paid for from the public purse. In the United States, the *Clinger–Cohen Act* (CCA) of 1996 (formerly known as the IT Management Reform Act) was enacted to require public

bodies to manage their information systems investment in a way that protected the interests of the taxpayers. The US Congress had been concerned about the effectiveness of investment in IT systems after a number of spectacular failures; the CCA was designed to prevent further failures.

The resulting pressure on public bodies, and particularly federal government departments, to conform to the requirements of the CCA resulted in the development of a number of enterprise architecture frameworks in the USA. The best known are the Federal Enterprise Architecture Framework (FEAF) and the Treasury Enterprise Architecture Framework (TEAF). There are no equivalent regulations that apply across the European Union, but a number of individual countries have developed frameworks for enterprise architecture, for example the German Standards and Architectures for eGovernment Applications (SAGA, 2003) and the United Kingdom's e-Government Interoperability Framework (e-GIF, 2004).

The Sarbanes–Oxley Act (SOX) was enacted in the USA following the collapse of Enron and a number of other large private-sector company failures. It imposes standards for financial recording, reporting and audit on US companies and overseas subsidiaries. Businesses have to be able to show that they are compliant with the requirements of SOX.

These legislative changes in the USA have sparked a worldwide concern with corporate governance. By governance we mean the decision-making processes, responsibilities and structures within an organization. In a world where information technology is pervasive in the financial accounting, control and management of both public and private bodies, the concept of governance extends into the way in which information systems are defined, developed, procured and managed.

Enterprise architectures provide a way of modelling the enterprise and aspects of the way it conducts business and of driving these concepts down into the practical issues of how the information systems are intended to support the business. Outside the world of US federal government, there is really only one significant approach to developing an enterprise architecture, and this is the Zachman framework (Zachman, 1987), developed originally by John Zachman, and extended in collaboration with John Sowa (Sowa and Zachman, 1992).

The Zachman framework seeks to build explicit models of the enterprise from two views. The first asks the questions What? How? Where? Who? When? and Why? The second looks at the system at different levels, from the most conceptual, business view down to the view of the actual implemented system. The two dimensions are usually viewed as a matrix and the values that fill the thirty-six cells in the matrix are the actual models of aspects of the enterprise at different levels and from different perspectives.

The task of the *enterprise architect* is to build up a total picture of the enterprise using these categories. This total picture of the enterprise and its systems supports the process of ensuring that any IT investment is aligned to the goals of the business. Clearly this is a daunting task for a large organization, and one of the criticisms of the Zachman framework is that it is a heavyweight approach to enterprise architecture.

In an organization that has adopted any kind of enterprise architecture framework, that framework should be the starting point for identifying constraints on the architecture of new systems.

12.4.3 Technical reference architectures

Whereas enterprise architectures address the entire organization and its systems, *technical reference architectures* focus on the technology that is used within the enterprise, the standards for the technologies to apply and guidance on how to apply that technology. This may be in terms of a standards document, or a list of approved technologies or architectural models that show how different technologies should be applied in a typical system.

For organizations that do not have the time or resources to develop their own framework for technology standards, The Open Group produced The Open Group Architecture Framework (TOGAF) in 1995 (The Open Group, 2002). The current version is 8.1.

TOGAF consists of three main parts.

- The Architecture Development Method describes an approach for developing enterprise IT architectures.
- The Enterprise Continuum shows the continuum of architectures:
  - Foundation Architectures, which consist of abstract building blocks that support all kinds of technical architectures
  - Common Systems Architectures, which provide models for typical domains based on the Foundation Architectures
  - Industry Architectures, which show how Common Systems Architectures are typically implemented in different types of industry
  - Organization Architectures, which address the specific architectural needs of particular organizations.
- The Resources section provides a range of useful information and examples of architectural patterns, principles and other guidance.

The Open Group also maintains an online Standards Information Base (SIB) that lists hundreds of IT standards categorized according to the building blocks in the Foundation Architectures model.

12.5 Architectural Styles

Architects designing buildings do not start from scratch every time they are given a new commission. They design buildings that are similar to others that they or other architects have built previously, and they learn what works and what does not. Systems architects are very similar: they design systems that conform to the prevailing standards, and fashions in systems architecture come and go, like flying buttresses on Gothic churches or lifts on the outside of buildings.

In systems architecture, the term *architectural styles* is used to apply to these ways of designing systems that conform to the prevailing fashion. Often these fashions are the result of changes in technology: until the advent of the PC, it would not have been possible to implement client–server system architectures using PCs connected to mini-computers. Architectural styles also apply to software architecture. Bass et al. (2003) describe five main types: independent components, data flow, data centred, virtual machine, and call and return, each with subtypes. Each style has characteristics that make it more or less suitable for certain types of application. We will consider some of the major alternatives. It is worth noting that

software architectures have been documented in the patterns form by Buschmann et al. (1996) and Schmidt et al. (2000) amongst others.

12.5.1 Subsystems

A subsystem typically groups together elements of the system that share some common properties. An object-oriented subsystem encapsulates a coherent set of responsibilities in order to ensure that it has integrity and can be maintained. For example, the elements of one subsystem might all deal with the human–computer interface, the elements of another might all deal with data management and the elements of a third may all focus on a particular functional requirement.

The subdivision of an information system into subsystems has the following advantages.

- It produces smaller units of development.
- It helps to maximize reuse at the component level.
- It helps the developers to cope with complexity.
- It improves maintainability.
- It aids portability.

Each subsystem should have a clearly specified boundary and fully defined interfaces with other subsystems. A specification for the interface of a subsystem defines the precise nature of the subsystem's interaction with the rest of the system but does not describe its internal structure (this is a high-level use of contracts, which are described in Chapter 10). A subsystem can be designed and constructed independently of other subsystems, simplifying the development process. Subsystems may correspond to increments of development that can be delivered individually as part of an incremental lifecycle (if the developers are using the spiral lifecycle model or an iterative and incremental approach such as the Unified Process).

Dividing a system into subsystems is an effective strategy for handling complexity. Sometimes it is only feasible to model a large complex system piece by piece, with the subdivision forced on the developers by the nature of the application. Splitting a system into subsystems can also aid reuse, as each subsystem may correspond to a component that is suitable for reuse in other applications. A judicious choice of subsystems during design can reduce the impact on the overall system of a change to its requirements. For example, consider an information system that contains a presentation subsystem that deals with the human–computer interface (HCI). A change to the data display format need not affect other subsystems. Of course there may still be some changes to the requirements that affect more than one subsystem. The aim is to localize the consequences of change, so that a change in one subsystem does not trigger changes in other subsystems (sometimes referred to as the ripple effect). Moving an application from one implementation platform to another can be much easier if the software architecture is appropriate. An example of this would be the conversion of a Windows application so that it could run in a Unix environment. This would require changes to the software that implements the human–computer interface. If this is dealt with by specialized subsystems then the overall software change is localized to these subsystems. As a result, the system as a whole is easier to port to a different operating environment.

Each subsystem provides services for other subsystems, and there are two different styles of communication that make this possible. These are known as *client–server* and *peer-to-peer* communication and are shown in Fig. 12.5.

Client–server communication requires the client to know the interface of the server subsystem, but the communication is only in one direction. The client subsystem requests services from the server subsystem and not vice versa. Peer-to-peer communication requires each subsystem to know the interface of the other, thus coupling them more tightly. The communication is two-way since either peer subsystem may request services from the other.

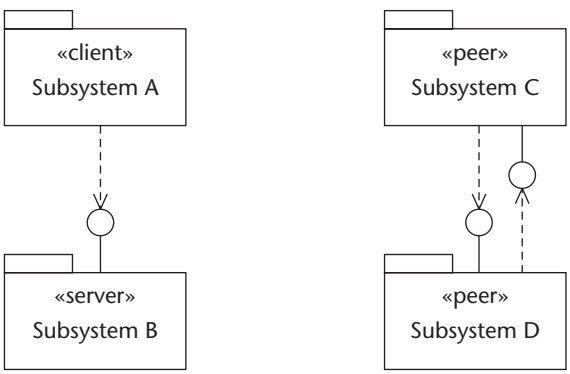
In general, client–server communication is simpler to implement and to maintain, as the subsystems are less tightly coupled than they are when peer-to-peer communication is used. In Fig. 12.5 the subsystems are represented using packages that have been stereotyped to indicate their role. Component and deployment diagrams can also be used to model the implementation of subsystems (see Chapter 20).

12.5.2 Layering and partitioning

There are two general approaches to the division of a software system into subsystems. These are known as *layering*—so called because the different subsystems usually represent different levels of abstraction[1]—and *partitioning*, which usually means that each subsystem focuses on a different aspect of the functionality of the system as a whole. In practice both approaches are often used together on one system, so that some of its subsystems are divided by layering, while others are divided by partitioning.

Layered subsystems

Layered architectures are among the most frequently used high-level structures for a system. A schematic of the general structure is shown in Fig. 12.6.

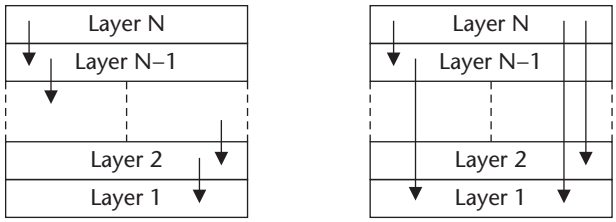


The server subsystem does not depend on the client subsystem and is not affected by changes to the client's interface

Each peer subsystem depends on the other and each is affected by changes in the other's interface

Figure 12.5 Styles of communication between subsystems.

1 Or layers of service.



Closed architecture– messages may only be sent to the adjacent lower layer

Open architecture– messages can be sent to any lower layer

Figure 12.6 Schematic of a layered architecture.

Each layer corresponds to one or more subsystems, which may be differentiated from each other by differing levels of abstraction or by a different focus of their functionality. It works like this: the top layer uses services provided by the layer immediately below it. This in turn may require the services of the next layer down. Layered architectures can be either open or closed, and each style has its particular advantages. In a closed layered architecture a certain layer (say layer N) can only use the services of the layer immediately below it (layer N – 1). In an open layered architecture layer N may directly use the services of any of the layers that lie below it.

A closed architecture minimizes dependencies between the layers and reduces the impact of a change to the interface of any one layer. An open layered architecture produces more compact code since the services of all lower level layers can be accessed directly by any layer above them without the need for extra program code to pass messages through each intervening layer. However, this breaks the encapsulation of the layers, increases the dependencies between layers and increases the difficulty caused when a layer needs to be changed.

Networking protocols provide some of the best known examples of layered architectures. A network protocol defines how computer programs executing on different computers communicate with each other. Protocols can be defined at various levels of abstraction and each level can be mapped onto a layer. The OSI (Open Systems Interconnection) Seven Layer Model was defined by the International Standardization Organization (ISO) as a standard architectural model for network protocols (Tanenbaum et al., 2002). The structure provides flexibility for change since a layer may be changed internally without affecting other layers, and it enables the reuse of layer components. The OSI Seven Layer Model is illustrated in Fig. 12.7.

Buschmann et al. (1996) suggest that a series of issues need to be addressed when applying a layered architecture in an application. These include:

- maintaining the stability of the interfaces of each layer
- the construction of other systems using some of the lower layers
- variations in the appropriate level of granularity for subsystems[2]

2 The context determines an appropriate size for the subsystems. Granularity refers to the size of the elements of a larger whole, fine-grained being small elements and coarse-grained being large.

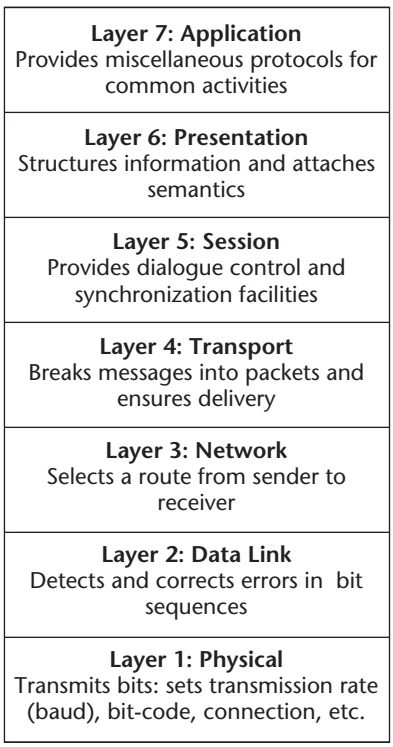


Figure 12.7 OSI Seven Layer Model (adapted from Buschmann et al., 1996).

- the further subdivision of complex layers
- performance reductions due to a closed layered architecture.

The OSI model has seven layers only because it covers every aspect of the communication between two applications, ranging from application-oriented processes to drivers and protocols that directly control network hardware devices. Many layered architectures are much simpler than this. Figure 12.8 shows a simple example of a three layer architecture.

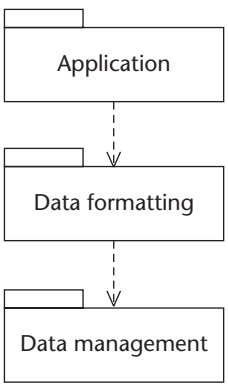


Figure 12.8 Simple layered architecture.

The lowest layer of the architecture in Fig. 12.8 consists of data management library classes. The layer immediately above this, the data formatting layer, uses services that are provided by the data management library classes in order to get data from a database management system. This data is formatted before it is passed upwards to the application layer. Supposing this system were to be modified to allow it to use a different database management system, the layered architecture limits major changes to the data management library class layer with some possible changes to the data formatting layer.

The following steps are adapted from Buschmann et al. (1996) and provide an outline process for the development of a layered architecture for an application. Note that this does not suggest that the specification of a system's architecture is a rule-based procedure. The steps offer guidelines on the issues that need to be addressed during the development of a layered architecture.

1. Define the criteria by which the application will be grouped into layers. A commonly used criterion is level of abstraction from the hardware. The lowest layer provides primitive services for direct access to the hardware while the layers above provide more complex services that are based upon these primitives. Higher layers in the architecture carry out tasks that are more complex and correspond to concepts that occur in the application domain.
2. Determine the number of layers. Too many layers will introduce unnecessary overheads while too few will result in a poor structure.
3. Name the layers and assign functionality to them. The top layer should be concerned with the main system functions as perceived by the user. The layers below should provide services and infrastructure that enable the delivery of the functional requirements.
4. Specify the services for each layer. In general it is better in the lower layers to have a small number of low-level services that are used by a larger number of services in higher layers.
5. Refine the layering by iterating through steps 1 to 4.
6. Specify interfaces for each layer.
7. Specify the structure of each layer. This may involve partitioning within the layer.
8. Specify the communication between adjacent layers (this assumes that a closed layer architecture is intended).
9. Reduce the coupling between adjacent layers. This effectively means that each layer should be strongly encapsulated. Where a client–server communication protocol will be used, each layer should have knowledge only of the layer immediately below it.

One of the simplest application architectures has only two layers—the application layer and a database layer. Tight coupling between the user interface and the data representation would make it more difficult to modify either independently, so a middle layer is often introduced in order to separate the conceptual structure of the problem domain. This gives the architecture shown in Fig. 12.9, which is commonly used for business-oriented information systems.

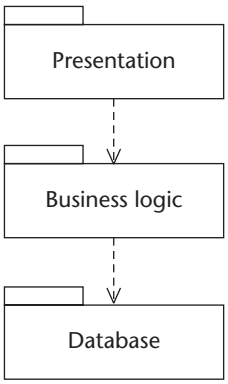


Figure 12.9 Three layer architecture.

A common four layer architecture separates the business logic layer into application logic and domain layers, and this is illustrated in Fig. 12.10. The approach that has been adopted during the analysis activity of use case realization results in the identification of boundary, control and entity classes. It is easy to see that it is possible to map the boundary classes onto a presentation layer, the control classes onto an application logic layer and the entity classes on a domain layer. Thus from an early stage in the development of an information system some element of layering is being introduced into the software architecture. However, it is important to appreciate that as we move through design, the allocation of responsibility amongst these types of class may be adjusted to accommodate non-functional requirements.

Separation of the application logic layer from the domain layer may be further justified because several applications share (or are likely to share) one domain layer, or because the complexity of the business objects forces a separation into two layers. It can also be used when the objects are physically distributed (see

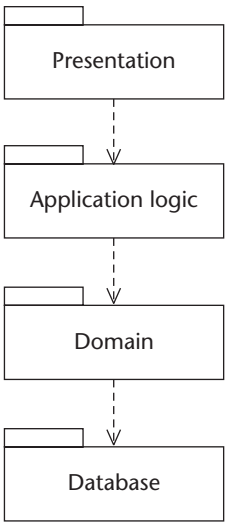


Figure 12.10 Four layer architecture.

Chapter 19). However, it must be emphasized that there is no perfect solution to this kind of design problem. There are only solutions that have different characteristics (perhaps different levels of efficiency or maintainability). A good design solution is one that balances competing requirements effectively.

Layered architectures are used quite widely. J2EE (Sun Java Centre, 2005) adopts a multi-tiered[3] approach and an associated patterns catalogue has been developed. The architecture has five layers (client, presentation, business, integration and resource tiers) and the patterns catalogue addresses the presentation, business and integration tiers.

Partitioned subsystems

As suggested earlier, some layers within a layered architecture may have to be decomposed because of their intrinsic complexity. Figure 12.11 shows a four layer architecture for part of Agate's campaign management system that also has some partitioning in the upper layers.

In this example the application layer corresponds to the analysis class model for a single application, and is partitioned into a series of subsystems. These subsystems are loosely coupled and each should deliver a single service or coherent group of services. The `Campaign Database` layer provides access to a database that contains all the details of the campaigns, their adverts and the campaign teams. The `Campaign Domain` layer uses the lower layer to retrieve and store data in the database and provides common domain functionality for the layers above. For example, the `Advert` subsystem might support individual advert costing while the `Campaign Costs` subsystem uses some of the same common domain functionality when costing a complete campaign. Each application subsystem has its own presentation layer to cater for the differing interface needs of different user roles[4].

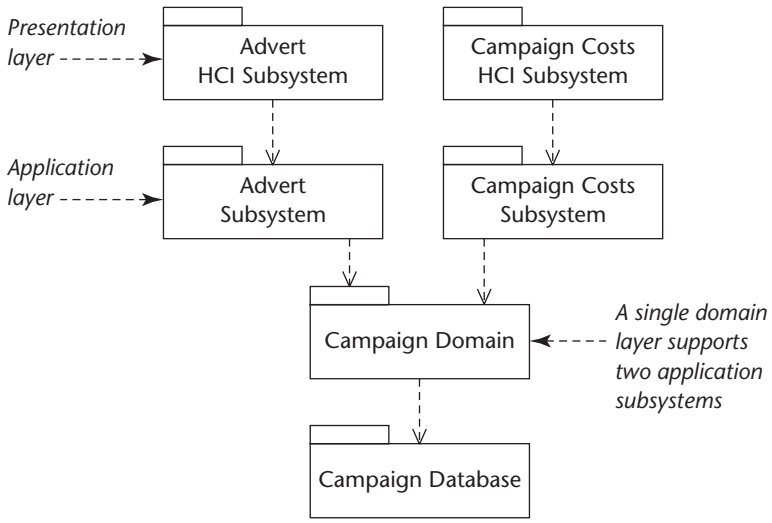


Figure 12.11 Four layer architecture applied to part of the Agate campaign management system.

3 These use the term tier as broadly equivalent to layer.

4 This example is for illustrative purposes only. Our analysis class model for Agate is too small to justify this kind of partitioning in practice.

A system may be split into subsystems during analysis because of the system's size and complexity. However, the analysis subsystems should be reviewed during design for coherence and compatibility with the overall system architecture.

The subsystems that result from partitioning should have clearly defined boundaries and well specified interfaces, thus providing high levels of encapsulation so that the implementation of an individual subsystem may be varied without causing dependent changes in the other subsystems. The process of identifying subsystems within a particular layer can be detailed in much the same way as for subsystem layers.

12.5.3 Model–View–Controller

Many interactive systems use the Model–View–Controller (MVC) architecture. This structure was first used with Smalltalk but has since become widely used with many other object-oriented development environments. The MVC architecture separates an application into three major types of component: models that comprise the main functionality, views that present the user interface and controllers that manage the updates to views. This structure is capable of supporting user requirements that are presented through differing interface styles, and it aids maintainability and portability.

It is common for the view of an information system that is required for each user to differ according to their role. This means that the data and functionality available to any user should be tailored to his or her needs. The needs of different types of user can also change at varying rates. For both these reasons it makes sense to give each user access to only the relevant part of the functionality of the system as a whole. For example, in the Agate case study many users need access to information about campaigns, but their perspectives vary. The campaign manager needs to know about the current progress of a campaign. She is concerned with the current state of each advertisement and how this impacts on the campaign as a whole—is it prepared and ready to run, or is it still in the preparation stage? If an advert is behind schedule, does this affect other aspects of the campaign? The graphic designer also needs access to adverts but he is likely to need access to the contents of the advert (its components and any notes that have been attached to it) as well as some scheduling information. A director may wish to know about the state of all live campaigns and their projected income over the next six months. This gives at least three different perspectives on campaigns and adverts, each of which might use different styles of display. The director may require charts and graphs that summarize the current position at quite a high level. The campaign manager may require lower level summaries that are both textual and graphical in form. The graphic designer may require detailed textual displays of notes with a capability to display graphical images of an advert's content. Ideally, if any information about a campaign or an advert is updated in one view then the changes should also be immediately reflected in all other views. Figure 12.12 shows a possible architecture, but some problems remain.

The design of such varied and flexible user interfaces that still incorporate the same core functionality is likely to be expensive because elements of functionality may have been duplicated for different interfaces. This makes the software more complex and thus also more error prone. There is an impact on maintainability too, since any change to core functionality will necessitate changes to each interface subsystem.

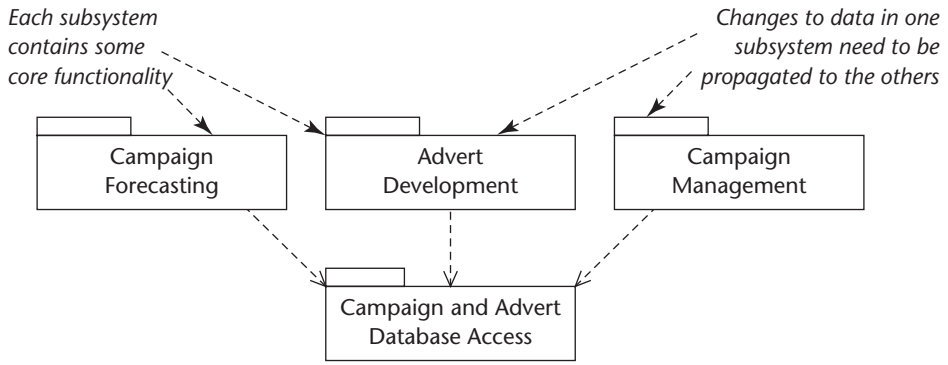


Figure 12.12 Multiple interfaces for the same core functionality.

We repeat below some of the difficulties that need to be resolved for this type of application.

- The same information should be capable of presentation in different formats in different windows.
- Changes made within one view should be reflected immediately in the other views.
- Changes in the user interface should be easy to make.
- Core functionality should be independent of the interface to enable multiple interface styles to co-exist.

While the four layer architecture in Fig. 12.11 resolves some of these problems it does not handle the need to ensure that all view components are kept up to date. The MVC architecture solves this through its separation of core functionality (model) from the interface and through its incorporation of a mechanism for propagating updates to other views. The interface itself is split into two elements: the output presentation (view) and the input controller (controller).

Figure 12.13 shows the basic structure of the MVC architecture.

The responsibilities of the components of an MVC architecture are listed below.

- **Model.** The model provides the central functionality of the application and is aware of each of its dependent view and controller components.
- **View.** Each view corresponds to a particular style and format of presentation of information to the user. The view retrieves data from the model and updates its presentations when data has been changed in one of the other views. The view creates its associated controller.
- **Controller.** The controller accepts user input in the form of events that trigger the execution of operations within the model. These may cause changes to the information and in turn trigger updates in all the views ensuring that they are all up to date.
- **Propagation Mechanism.** This enables the model to inform each view that the model data has changed and as a result the view must update itself. It is also often called the dependency mechanism.

Figure 12.14 represents the capabilities offered by the different MVC components as they might be applied to part of the campaign management system at Agate.

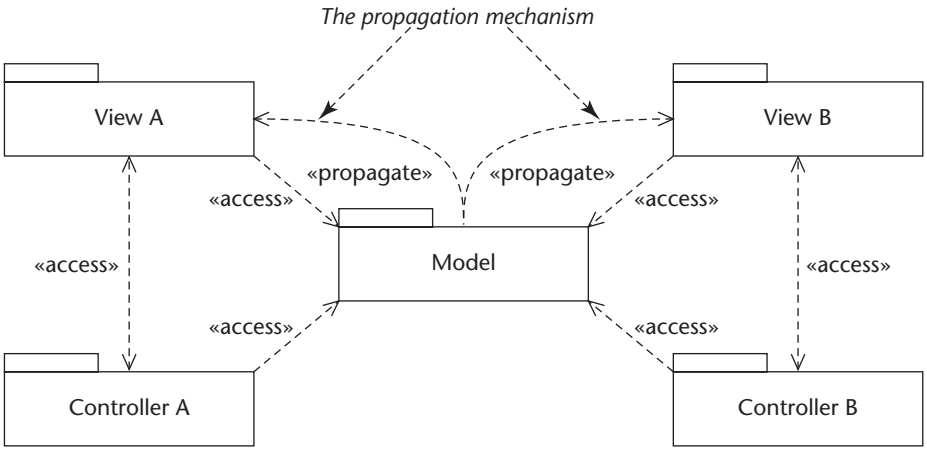


Figure 12.13 General structure of Model–View–Controller (adapted from Hopkins and Horan, 1995).

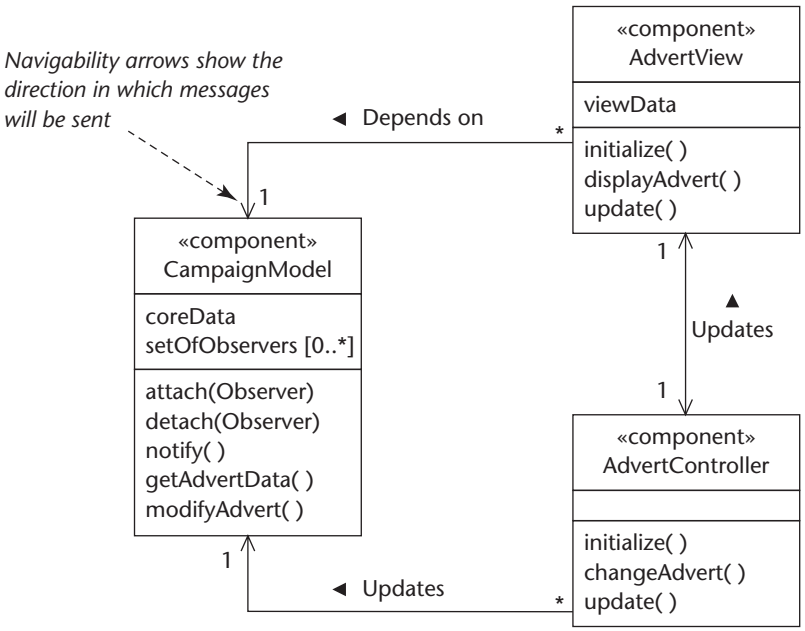


Figure 12.14 Responsibilities of MVC components, as applied to Agate.

The operation `update()` in the `AdvertView` and `AdvertController` components triggers these components to request data from the `CampaignModel` component[5]. This model component has no knowledge of the way that each view and controller component will use its services. It need only know that all view and controller components must be informed whenever there is a change of state (a modification either of object attributes or of their links).

5 In this example the `CampaignModel` will hold details of campaigns and their adverts.

The `attach()` and `detach()` services in the `CampaignModel` component enable views and controllers to be added to the `setOfObservers`. This contains a list of all components that must be informed of any change to the model core data. In practice there would be separate views, each with its own controller, to support the requirements of the campaign manager and the director.

The interaction sequence diagram in Fig. 12.15 illustrates the communication that is involved in the operation of an MVC architecture. (The choice of message type–synchronous or asynchronous–shown in this diagram is only one of the possibilities that could be appropriate, the features of the implementation environment would influence the actual design decision.) An `AdvertController` component receives the interface event `changeAdvert`. In response to this event the controller invokes the `modifyAdvert` operation in the `CampaignModel` object. The execution of this operation causes a change to the model.

For example, the target completion date for an advertisement is altered. This change of state must now be propagated to all controllers and views that are currently registered with the model as active. To do this the `modifyAdvert` operation invokes the `notify` operation in the model, which sends an `update` message to the view. The view responds to the `update` message by executing the `display Advert` operation which requests the appropriate data from the model via the `getAdvertData` operation. The model also sends an `update` message to the `AdvertController`, which then requests the data it needs from the model.

One of the most important aspects of the MVC architecture is that each model knows only which views and controllers are registered with it, but not what they do. The `notify` operation causes an update message to all the views and

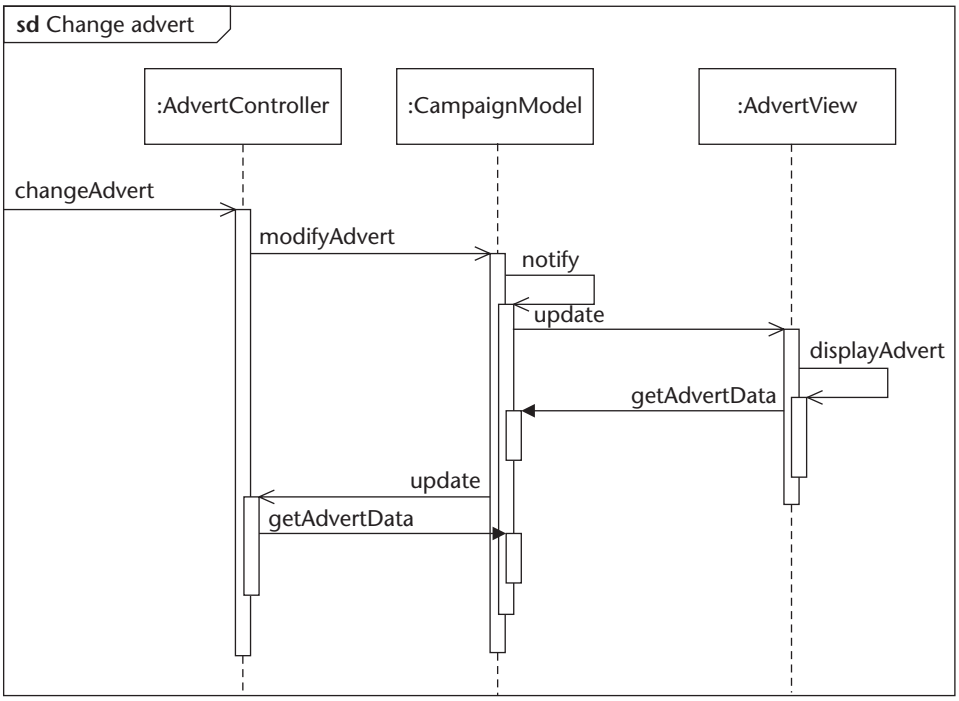


Figure 12.15 MVC component interaction.

controllers (for clarity, only one view and one controller are shown in the diagram, but interaction with the others would be similar). The `update` message from the model is in effect saying to the views and controllers 'I have been updated and you must now ensure that your data is consistent'. Thus the model, which should be the most stable part of the application, is unaffected by changes in the presentation requirements of any view or controller. The change propagation mechanism can be structured so that further views and controllers can be added without causing a change to the model. Each of these may support different interface requirements but require the same model functionality. However, since views and controllers need to know how to access the model in order to get the information they require, some changes in the model will inevitably still cause changes in other components.

Other kinds of communication may take place between the MVC components during the operation of the application. The controller may receive events from the interface that require a change in the way that some data is presented to the user but do not cause a change of state. The controller's response to such an event would be to send an appropriate message to the view. There would be no need for any communication with the model.

12.5.4 Architectures for distributed systems

Distributed information systems have become more common as communications technology has improved and have also become more reliable. An information system may be distributed over computers at the same location or at different locations. Since Agate has offices around the world, it may need information systems that use data that is distributed among different locations. If Agate grows, it may also open new offices and require new features from its information systems. An architecture that is suitable for distributed information systems needs also to be flexible so that it can cope with change. A distributed information system may be supported by software products such as distributed database management systems or object request brokers or may adopt a service-oriented architecture (these are discussed in Chapter 19).

A general *broker* architecture for distributed systems is described by Buschmann et al. (1996). A simplified version of the broker architecture is shown in Fig. 12.16.

A broker component increases the flexibility of the system by decoupling the client and server components. Each client sends its requests to the broker rather

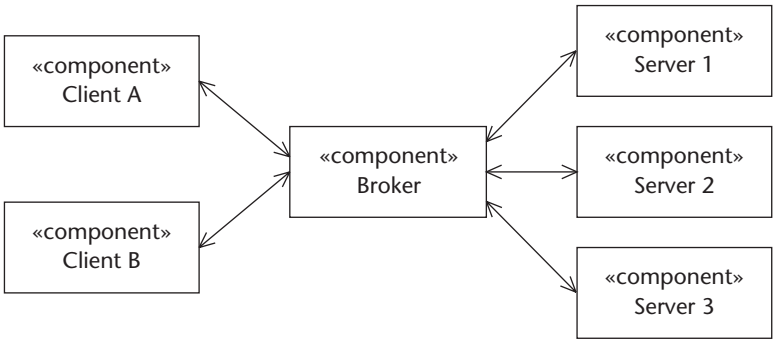


Figure 12.16 Schematic of simplified broker architecture.

than communicating directly with the server component. The broker then forwards the service request to an appropriate server. A broker may offer the services of many servers and part of its task is to identify the relevant server to which a service request should be forwarded. The advantage offered by a broker architecture is that a client need not know where the service is located, and it may therefore be deployed on either a local or a remote computer. Only the broker needs to know the location of the servers that it handles.

Figure 12.17 shows a sequence diagram for client–server communication using the broker architecture. The diagram is drawn with asynchronous message types but the actual implementation may involve both synchronous and asynchronous message types. In this example the server subsystem is on a local computer. In addition to the broker itself, two additional *proxy* components have been introduced to insulate the client and server from direct access with the broker. On the client side a `ClientSideProxy` receives the initial request from the client and packs the data in a format suitable for transmission. The request is then forwarded to the `Broker` which finds an appropriate server and invokes the required service via the `ServerSideProxy`.

The `ServerSideProxy` then unpacks the data and issues the service request sending the `service` message to the `Server` object. The `service` operation then executes and on completion responds to the `ServerSideProxy`. The response is then sent to the `Broker` which forwards it to the originating `ClientSideProxy`.

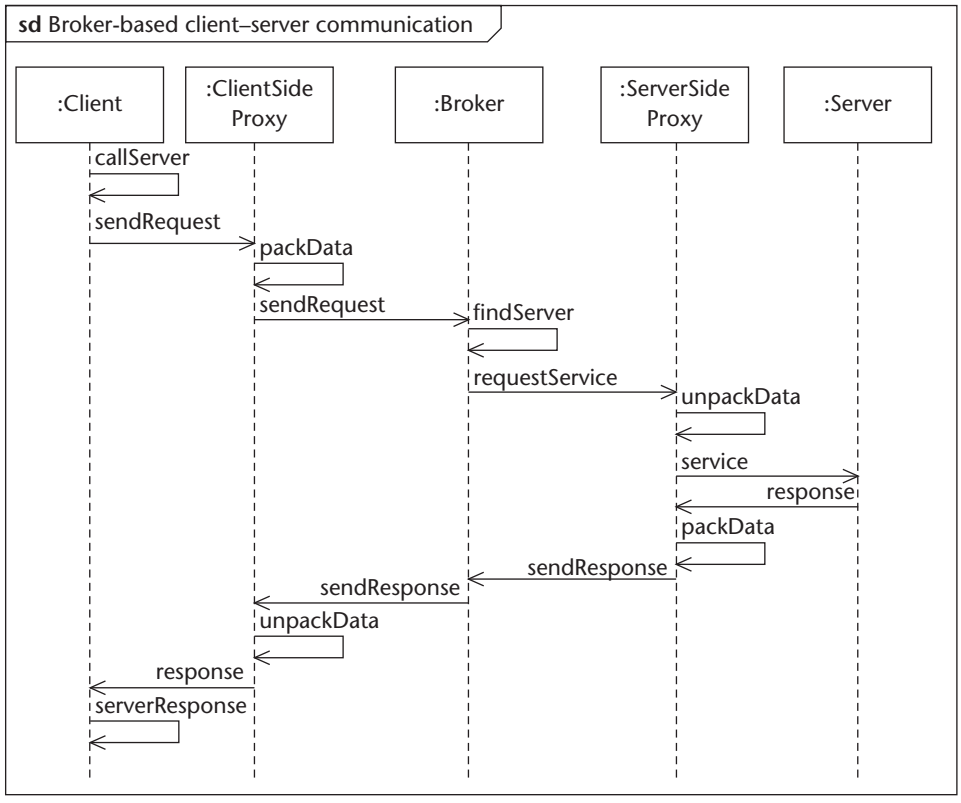


Figure 12.17 Broker architecture for local server (adapted from Buschmann et al., 1996).

Note that these are both new messages and not returns. The reason for this is that a broker does not wait for each response before handling another request. Once its `sendRequest` activation has been completed, the broker will in all probability deal with many other requests and thus requires a new message from the `ServerSideProxy` object to cause it to enter a new activation. Unlike the broker, the `ClientSideProxy` has remained active; this then unpacks the message and the response becomes available to the `Client` as control returns.

Figure 12.18 shows how the participants in this interaction can be allocated to different processes, with the client and its proxy running in one process thread, the broker in another and the server and its proxy in a third.

Figure 12.19 shows a schematic broker architecture that uses *bridge* components to communicate between two remote processors. Each bridge converts service requests into a network specific protocol so that the message can be transmitted. Figure 12.20 shows a possible allocation of these components to processes.

12.5.5 Organization structures for architecture and development

Dividing a system into subsystems has benefits for project management. Each subsystem can be allocated to a single development team, which can operate independently of other teams, provided that they adhere to the interface requirements for their subsystem. Where a subsystem must be split between two development teams, there is a heavy communications overhead that is incurred in ensuring that the different parts of the subsystem are constructed to consistent standards. In such cases the structure of either the organization or of the software tends to change so that they become more closely aligned with each other; this helps to minimize the communications overhead and is sometimes known as Conway's Law[6] (Coplien,

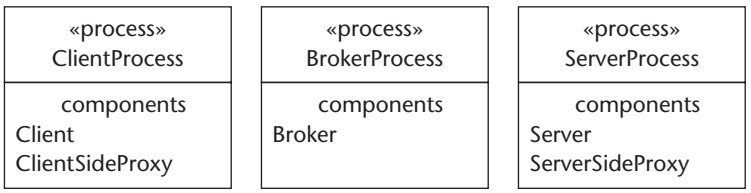


Figure 12.18 Process allocation of components in Figure 12.17.

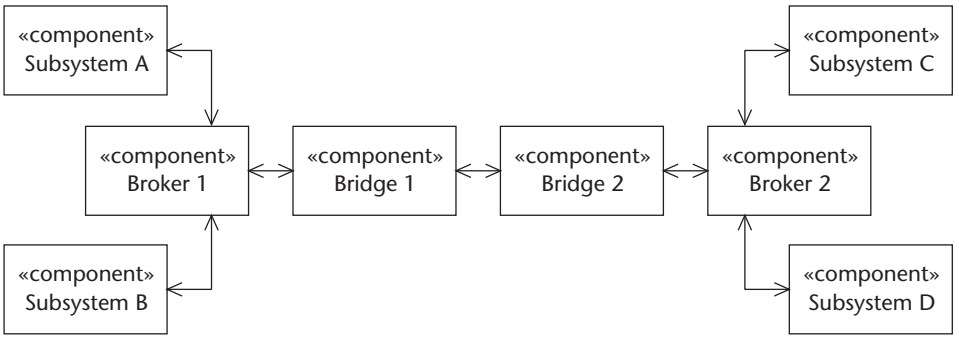


Figure 12.19 Schematic of broker architecture using bridge components.

6 This is an example of an organizational pattern.

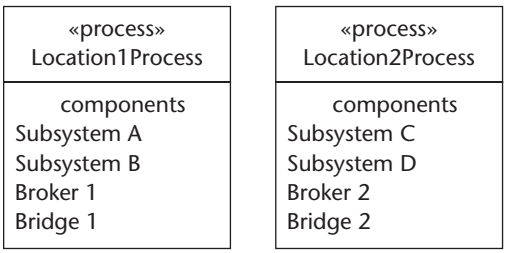


Figure 12.20 Process allocation of components in Figure 12.19.

1995). If a subsystem that is being developed by more than one team is cohesive, and the way it is split between teams has no apparent functional basis, then the teams may coalesce in practice and operate as one. Teams that are working on the same subsystem are sometimes inhibited from merging, perhaps because they are located on different continents. The subsystem should then be treated as if it were two separate subsystems. An interface between these two new subsystems can be defined and the teams can then operate autonomously. Where the allocation of one subsystem to two teams is such that one team deals with one set of requirements and the other deals with a different set of requirements, the subsystem can also be treated as if it were actually two subsystems, with a defined interface between them.

12.6 Concurrency

In most systems there are many objects that do not need to operate concurrently but some may need to be active simultaneously. Object-oriented modelling captures any inherent concurrency in the application principally through the development of interaction diagrams and state machines. The examination of use cases also helps with the identification of concurrency. There are several ways of using these models to identify circumstances where concurrent processing may be necessary. First, a use case may indicate a requirement that the system should be able to respond simultaneously to different events, each of which triggers a different thread of control. Second, if a state machine reveals that a class has complex nested states which themselves have concurrent substates, then the design must be able to handle this concurrency. The state machine for the class `Campaign` has nested concurrent states within the `Active` state (see Fig. 11.20) and there may be the possibility of concurrent activity. In this particular example, the concurrent activity that occurs in the real world need not necessarily be represented as concurrent processing in the computerized information system.

In cases where an object is required to exhibit concurrent behaviour it is sometimes necessary to split the object into separate objects in order to avoid the need for concurrent activity within any one object. Concurrent processing may also be indicated if interaction diagrams reveal that a single thread of control requires that operations in two different objects should execute simultaneously, perhaps because of asynchronous invocation. This essentially means that one thread of control is split into two or more active threads. An example of this is shown in Fig. 12.21.

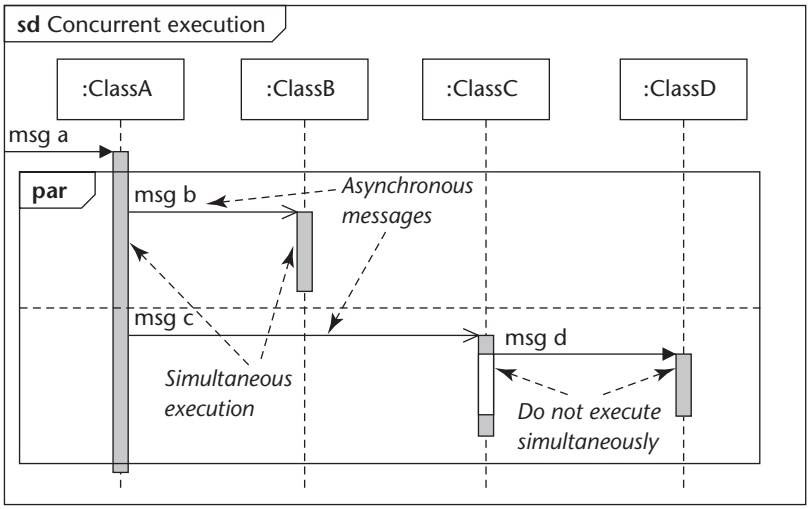


Figure 12.21 Concurrent activity in an interaction diagram.

Different objects that are not active at the same time can be implemented on the same logical processor (and thus also on the same physical processor—this distinction is explained below). Objects that must operate concurrently must be implemented on different logical processors (though perhaps still on the same physical processor).

The distinction between logical and physical concurrency is as follows. There are a number of ways of simulating the existence of multiple processors using only a single physical processor. For example, some operating systems (Unix and Windows XP) allow more one than one task to appear to execute at the same time, and are thus called multi-tasking operating systems. In fact, only one task really takes place at any one time, but the operating system shares the processor between different tasks so quickly that the tasks appear to execute simultaneously. Where there are no tight time constraints a multi-tasking operating system can provide a satisfactory implementation of concurrency. But it is important to ensure that the hardware configuration of the computer can cope with the demands of multi-tasking.

When there are tight time constraints a scheduler subsystem can be introduced that ensures that each thread of control operates within the constraints on its response time. Figure 12.22 illustrates a possible relationship between a scheduler and the other parts of a system. Events that are detected by the I/O (input/output) subsystems generate interrupts in the scheduler. The scheduler then invokes the appropriate thread of control. Further interrupts may invoke other threads of control and the scheduler allocates a share of physical processor time to each thread.

Another way of implementing concurrency is to use a multi-threaded programming language (such as Java). These permit the direct implementation of concurrency within a single processor task. Finally, a multi-processor environment allows each concurrent task to be implemented on a separate processor.

Most concurrent activity in a business information system can be supported by a multi-user environment. These are designed to allow many users to perform tasks

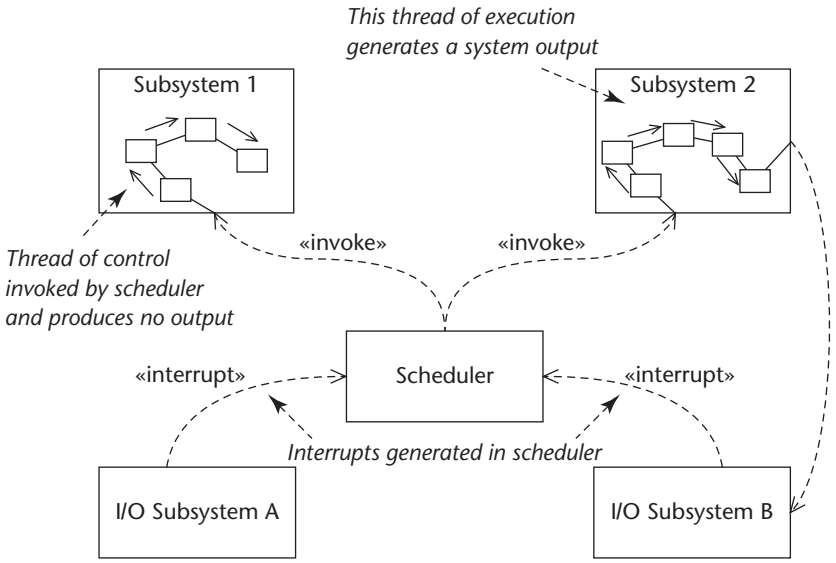


Figure 12.22 Scheduler handling concurrency.

simultaneously. Multi-user concurrent access to data is normally handled by a separate database management system (DBMS)—these are introduced briefly in Section 13.7 and are discussed in more detail in Chapter 17.

12.7 Processor Allocation

In the case of a simple, single-user system it is almost always appropriate for the complete system to operate on a single computer. The software for a multi-user information system (all or part of it) may be installed on many computers that use a shared database server. More complex applications sometimes require the use of more than one type of computer, where each provides a specialized kind of processing capability for a specific subsystem. An information system may also be partitioned over several processors, either because subsystems must operate concurrently or because some parts of the application need to operate in different locations (in other words, it is a distributed system). Information systems that use the Internet or company intranets for their communications are now widespread. Such distributed information systems operate on diverse computers and operating systems.

The allocation of a system to multiple processors on different platforms involves the following steps.

- The application should be divided into subsystems.
- Processing requirements for each subsystem should be estimated.
- Access criteria and location requirements should be determined.
- Concurrency requirements for the subsystems should be identified.
- Each subsystem should be allocated to an operating platform—either general purpose (PC or workstation) or specialized (embedded micro-controller or specialist server).

- Communication requirements between subsystems should be determined.
- The communications infrastructure should be specified.

The estimation of processing requirements requires careful consideration of such factors as event response times, the data throughput that is needed, the nature of the I/O that is required and any special algorithmic requirements. Access and location factors include the difficulties that may arise when a computer will be installed in a harsh operating environment such as a factory shop floor.

12.8 Agate Software Architecture

In the case study chapters, A2, A3 and A4, we have developed the models for the Agate system. The initial package architecture was shown in Fig. A2.8. However, this does not reflect a proper layering or partitioning of the software architecture. We may begin with a four layer architecture that separates responsibility for the user interface, the application logic, the domain classes and the database. A simple view of this is shown in Fig. 12.23.

However, we know that Agate requires the system to be capable of distribution. One option would be to adopt a thin-client architecture. In this approach, all four of the layers shown in Fig. 12.23 would be located on one or more servers, and the user interface would be generated as HTML and displayed in a browser. However, this would not give us the kind of interactivity that we have been modelling in our prototype user interfaces, which rely on a client program running on the users' PCs. So we need to decide where to split the system between the client side and the server side. The `Agate Control` package could be split into a client-side package that co-ordinates the user interface, playing the role of Controller, and a server-side package that orchestrates the business logic of the use cases and interacts with the domain classes. If we adopt this approach, we will break the closed layering of

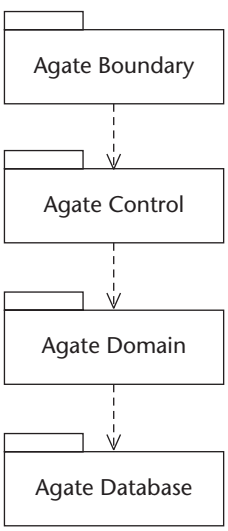


Figure 12.23 Four layer architecture for Agate.

the architecture of Fig. 12.23. Both the client-side and the server-side classes will need to understand the structure of the entity objects in the domain package (`Advert`, `Campaign`, `Client` etc.). If we develop in Java, the jar file containing these classes will need to be located on the client as well as the server, even if their operations are not invoked on the client. One way to reduce this dependency is to use lightweight versions of the entity classes in the `Agate Domain` package. These are classes that have the attributes of the entity classes, but do not have any operations apart from constructors and those operations necessary to get and set attribute values. This is an established pattern in J2EE systems, and is shown in Fig. 12.24.

Note that the dependency between the `Agate Client Control` package and the `Agate Value Objects` package is no more than that, a dependency. It does

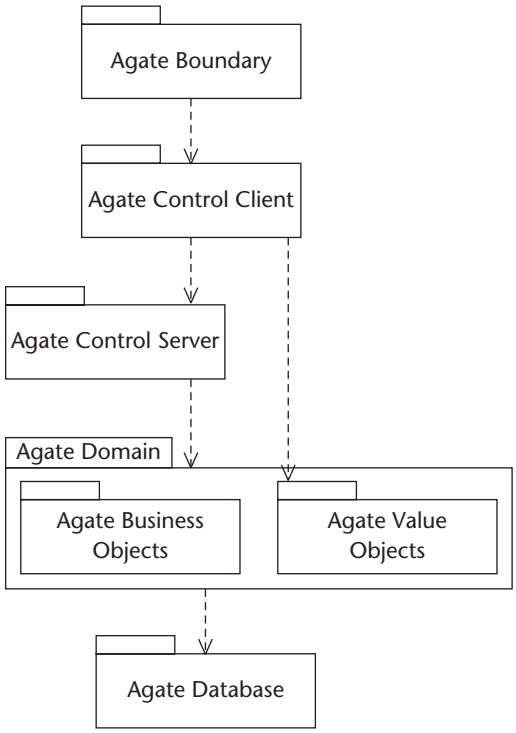


Figure 12.24 Package architecture for Agate.

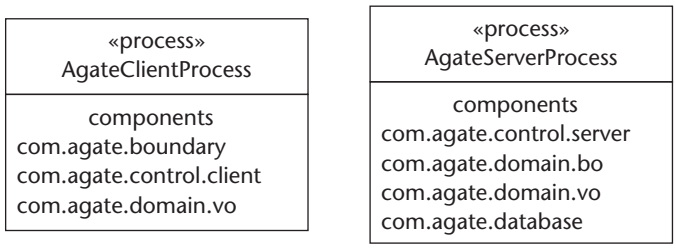


Figure 12.25 Process allocation for Agate.

not imply that there is some kind of communication across the network between the two. In fact if we implement the packages as Java packages, and deploy them, the value object package (`com.agate.domain.vo`) will exist in both the client process and the server process. This is shown in Figure 12.25.

We shall revisit this architecture in the case study chapter A5, once we have considered in more detail the design of classes, the user interface and the database.

12.9 Summary

Systems and software architecture have aspects in common with the architecture of buildings, and architectural models are typically produced using different views, which address different aspects of the architecture. In information systems, 'architecture is the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution' (IEEE, 2000). Many architects now use UML in order to produce architectural models of systems.

One of the key concerns of architects is to ensure that the architecture of the system will enable it to meet the quality attributes (non-functional requirements) that are expected of it. The models allow them to reason about how well the proposed structures and relationships will support demands on performance, reliability, reusability and other quality attributes.

The architecture of new systems is often constrained by existing systems, because they define either explicitly or implicitly the way in which systems are built within the organization, or because the new systems will have to inter-operate with the old. There is a growing interest in wrapping up existing systems as services to support a service-oriented architecture, or in extracting the business logic from heritage systems using reverse-engineering to produce platform-independent models, and then deriving new implementations in more modern technologies from the models. The Model-Driven Architecture movement places UML at the centre of its work and many of the features that have been improved or added in UML 2.0 are there to support MDA.

Large organizations may mandate approaches to architecture development based on enterprise architecture or technical reference architectures that lay down models of the business and how it operates (in the former case) or of standard technologies to be applied (in the latter case). Experienced architects also draw on architectural styles, which act as architectural patterns and provide well understood ways of constructing the high-level architecture of new systems.

Review Questions

12.1 Give a definition of architecture in an information systems context.

12.2 What is the difference between an architectural view and an architectural viewpoint?

12.3 What are the 4 + 1 views of architecture in the Unified Process?

12.4 What are the benefits of adopting an architecture-centric approach?

12.5 How do existing systems influence the architecture of new systems in the same organization?

12.6 Explain the difference between a PIM and a PSM.

12.7 What is meant by enterprise architecture?

12.8 What are the advantages of dividing a system into a collection of subsystems?

12.9 What is the difference between client–server and peer-to-peer communication between subsystems?

12.10 Why is an open layered architecture more difficult to maintain?

12.11 What are the disadvantages of the closed layered architecture?

12.12 What advantages would there be if the `Advert HCI` subsystem in Fig. 12.11 were designed to have direct access to the `Campaign Database` layer?

12.13 What are the main differences between the MVC architecture and the layered and partitioned architecture?

12.14 In what sense does a broker decouple two subsystems that need to communicate with each other? How does this work?

12.15 How do architectural divisions of systems help with project management?

12.16 Why is it sometimes necessary to design information systems that have explicitly concurrent behaviour?

12.17 How should you go about allocating system tasks to processors?

Case Study Work, Exercises and Projects

12.A Compare Soni's four aspects with the UP 4 + 1 views. What do they have in common and how do they differ?

12.B Consider a system that you use regularly. What, if anything, can you tell about the architecture of the system from the user's perspective?

12.C Develop a series of steps for the identification of partitioned subsystems within a layer in a layered architecture. Use the process for the identification of layers described in Section 12.5.2 as a starting point. Highlight any significant differences that you feel exist between the two processes.

12.D Investigate a framework for enterprise architecture. What support is there for it in modelling tools?

12.E Suggest a suitable layered architecture with any necessary partitioning for the FoodCo case study by following the procedures defined above.

Further Reading

Bass et al. (2003) is an updated version of their 1998 book, in which they have adopted UML as their architecture description language. The book mixes case studies from different kinds of projects with theory and practical guidance.

Garland and Anthony (2003) provide an excellent and practical approach to developing software architectures using UML.

Buschmann et al. (1996, 2000) provide further details of the architectures discussed in this chapter and describe other interesting alternatives.

Details of the OMG's MDA initiative can be found at www.omg.org/mda/.

The Zachman Institute for Framework Advancement (www.zifa.com) provides information about the Zachman framework.