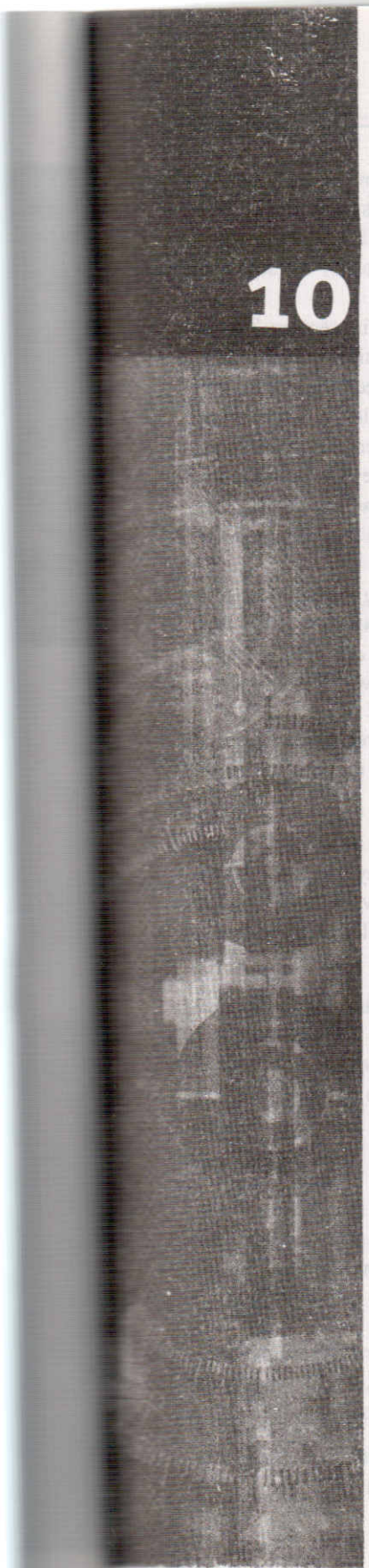
PART THREE

Design



10 Architectural design

Objectives

The objective of this chapter is to introduce the concepts of software architecture and architectural design. When you have read the chapter, you will

- understand why the architectural design of software is important;
- understand that different models may be required to document a system architecture;
- have been introduced to a number of different types of software architecture covering system structure, control and modular decomposition;
- understand how domain-specific architectural models may be used as a basis for product-line architectures and to compare architectural implementations.

Contents

Large systems are always decomposed into sub-systems that provide some related set of services. The initial design process of identifying these sub-systems and establishing a framework for sub-system control and communication is called *architectural design* and the output of this design process is a description of the *software architecture*.

The overall structure of the design process was discussed in section 3.4. In Figure 3.9, I presented a model of the design process where architectural design is the first stage in that process and represents a critical link between the design and requirements engineering processes. Ideally, a specification should not include any design information. In practice, this is unrealistic except for very small systems. Architectural decomposition is necessary to structure and organise the specification. A good example of this was introduced in Figure 2.4 which shows the architecture of an air traffic control system. The architectural model is often the starting point for the specification of the various parts of the system.

The architectural design process is concerned with establishing a basic structural framework for a system. It involves identifying the major components of the system and the communications between these components. Bass *et al.* (1998) discuss three advantages of explicitly designing and documenting a software architecture:

1. *Stakeholder communication* The architecture is a high-level presentation of the system that may be used as a focus for discussion by a range of different stakeholders.
2. *System analysis* Making the system architecture explicit at an early stage in the system development means that some analysis may be carried out. Architectural design decisions have a profound effect on whether or not the system can meet critical requirements such as performance, reliability and maintainability.
3. *Large-scale reuse* A system architecture is a compact, manageable description of how a system is organised and how the components interoperate. The architecture can be transferred across systems with similar requirements and so can support large-scale software reuse. It may be possible to develop product-line architectures where the same architecture is used across a range of related systems. I discuss this in section 10.4 and in Chapter 14.

Different designers approach the architectural design process in different ways. The process used depends on application knowledge and on the skill and intuition of the system architect. However, the following activities are common to all architectural design processes:

1. *System structuring* The system is structured into a number of principal sub-systems where a sub-system is an independent software unit. Communications between sub-systems are identified. This is covered in section 10.1.
2. *Control modelling* A general model of the control relationships between the parts of the system is established. This is covered in section 10.2.

3. *Modular decomposition* Each identified sub-system is decomposed into modules. The architect must decide on the types of module and their interconnections. This is covered in section 10.3.

These activities are usually interleaved rather than carried out in sequence. During any of these processes, you may have to develop the design in more detail to find out if architectural design decisions allow the system to meet its requirements.

There is no clear distinction between sub-systems and modules but I find it useful to think of them as follows:

1. *A sub-system* is a system in its own right whose operation does not depend on the services provided by other sub-systems. Sub-systems are composed of modules and have defined interfaces which are used for communication with other sub-systems.
2. *A module* is normally a system component that provides one or more services to other modules. It makes use of services provided by other modules. It is not normally considered to be an independent system. Modules are usually composed from a number of other, simpler system components.

The output of the architectural design process is an architectural design document. This consists of a number of graphical representations of the system models along with associated descriptive text. It should describe how the system is structured into sub-systems and how each sub-system is structured into modules. The different graphical models of the system present different perspectives on the architecture. Architectural models that may be developed may include:

1. A static structural model that shows the sub-systems or components that are to be developed as separate units.
2. A dynamic process model that shows how the system is organised into processes at run-time. This may be different from the static model.
3. An interface model that defines the services offered by each sub-system through their public interface.
4. Relationship models that show relationships such as data flow between the sub-systems.

A number of researchers have proposed the use of architectural description languages (ADLs) to describe system architectures. Bass *et al.* (1998) describe the main features of these languages. The basic elements of ADLs are components and connectors and they include rules and guidelines for well-formed architectures. However, like all specialised languages, they suffer from the disadvantage that they can only be understood by language experts – they are inaccessible to domain and application specialists. This makes them difficult to analyse from a practical perspective. I therefore think it likely they will only be used in a small number of applications. Rather, informal models and notations such as the UML will mostly be used for architectural description.

Architectural design may be based on a particular architectural model or style (Garlan and Shaw, 1993). An awareness of these models, their applications, strengths and weaknesses is important. I describe several different structural models, control models and decomposition models in this chapter.

However, the architectures of most large systems do not conform to a single model. Different parts of the system may be designed using different architectural models. Furthermore, in some cases, the system architecture may be a composite architecture. This is created by combining different architectural models. Designers must find the most appropriate model, then modify it according to the problem requirements. An example of this is shown in section 10.4 in the discussion of compiler architecture where a repository model is combined with a data-flow model.

The system architecture affects the performance, robustness, distributability and maintainability of a system. The particular style and structure chosen for an application may therefore depend on the non-functional system requirements:

1. *Performance* If performance is a critical requirement, this suggests that the architecture should be designed to localise critical operations within a small number of sub-systems with as little communication as possible between these sub-systems. This may mean using relatively large-grain rather than fine-grain components to reduce component communications.
2. *Security* If security is a critical requirement, this suggests that a layered structure for the architecture should be used with the most critical assets protected in the innermost layers and with a high level of security validation applied to these layers.
3. *Safety* If safety is a critical requirement, this suggests that the architecture should be designed so that safety-related operations are all located either in a single sub-system or in a small number of sub-systems. This reduces the costs and problems of safety validation and makes it possible to provide related protection systems.
4. *Availability* If availability is a critical requirement, this suggests that the architecture should be designed to include redundant components so that it is possible to replace and update components without stopping the system. Fault-tolerant system architectures for high-availability systems are covered in Chapter 18.
5. *Maintainability* If maintainability is a critical requirement, this suggests that the system architecture should be designed using fine-grain, self-contained components that may readily be changed. Producers of data should be separated from consumers and shared data structures should be avoided.

Obviously there is potential conflict between some of these architectures. For example, performance is improved by using large-grain components and maintainability by using fine-grain components. If both of these are important system requirements, some compromise solution must be found. As discussed above, this

can sometimes be achieved by using different architectural styles for different parts of the system.

10.1 System structuring

The first phase of the architectural design activity is usually concerned with decomposing a system into a set of interacting sub-systems. At its most abstract level, an architectural design may be depicted as a block diagram where each box in the diagram represents a sub-system. Boxes within boxes indicate that the sub-system has itself been decomposed to sub-systems. Arrows mean that data and/or control is passed from sub-system to sub-system in the direction of the arrows. An architectural block diagram presents an overview of the system structure. It is generally understandable to the various engineers who may be involved in the system development process.

Figure 10.1 is a structural model of the architecture for a packing robot system. This robotic system can pack different kinds of object. It uses a vision sub-system to pick out objects on a conveyor, identifies the type of object and selects the right kind of packaging. It then moves objects from the delivery conveyor to be packaged. Packaged objects are placed on another conveyor. Other examples of architectural designs at this level are shown in Figures 2.2 and 2.4.

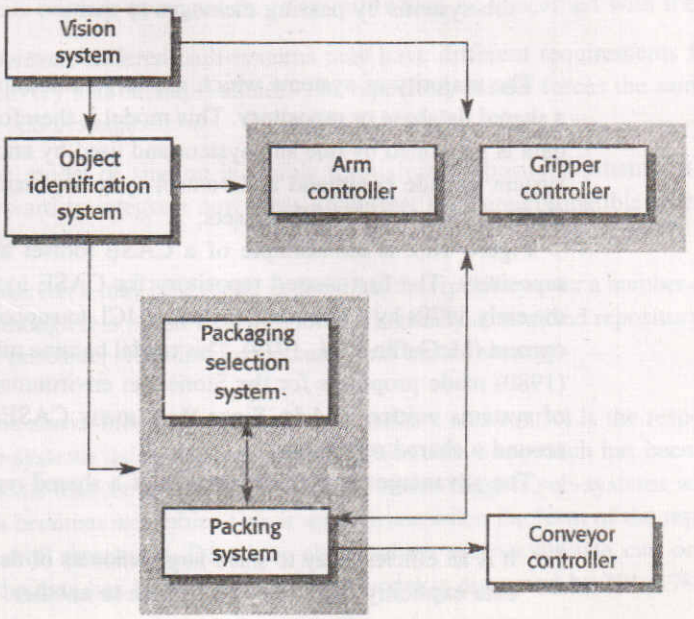


Figure 10.1 Block diagram of a packing robot control system

Bass *et al.* (1998) claim that simple box and line diagrams are not useful architectural representations as they do not show the nature of the relationships between system components nor their externally visible properties. From a software designer's perspective, this is absolutely correct. However, this type of model is effective for communication with system stakeholders and for project planning. As it is not cluttered with detail, stakeholders can relate to it and understand an abstract view of the system. It identifies the key sub-systems that are independently developed so managers can start assigning people to plan the development of these systems. Box and line diagrams should certainly not be the only architectural representation that is used; however, I think that they are one of a number of useful architectural models.

More specific models of the structure may be developed which show how sub-systems share data, how they are distributed and how they interface with each other. I discuss three of these standard models, namely a repository model, a client–server model and an abstract machine model in this section.

10.1.1 The repository model

Sub-systems making up a system must exchange information so that they can work together effectively. There are two fundamental ways in which this can be done.

1. All shared data is held in a central database that can be accessed by all sub-systems. A system model based on a shared database is sometimes called a *repository model*.
2. Each sub-system maintains its own database. Data is interchanged with other sub-systems by passing messages to them.

The majority of systems which use large amounts of data are organised around a shared database or repository. This model is therefore suited to applications where data is generated by one sub-system and used by another. Examples of this type of system include command and control systems, management information systems, CAD systems and CASE toolsets.

Figure 10.2 is an example of a CASE toolset architecture based on a shared repository. The first shared repository for CASE tools was probably developed in the early 1970s by a UK company called ICL to support their operating system development (McGuffin *et al.*, 1979). This model became more widely known when Buxton (1980) made proposals for the Stoneman environment to support the development of systems written in Ada. Since then, many CASE toolsets have been developed around a shared repository.

The advantages and disadvantages of a shared repository are as follows:

1. It is an efficient way to share large amounts of data. There is no need to transmit data explicitly from one sub-system to another.

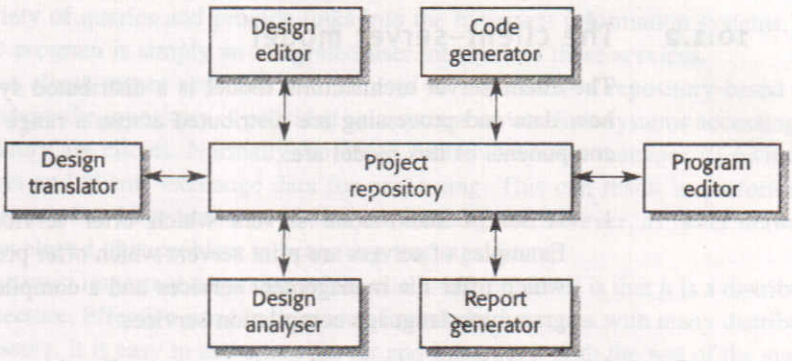


Figure 10.2 The architecture of an integrated CASE toolset

2. However, sub-systems must agree on the repository data model. Inevitably, this is a compromise between the specific needs of each tool. Performance may be adversely affected by this compromise. It may be difficult or impossible to integrate new sub-systems if their data models do not fit the agreed schema.
3. Sub-systems which produce data need not be concerned with how that data is used by other sub-systems.
4. However, evolution may be difficult as a large volume of information is generated according to an agreed data model. Translating this to a new model will certainly be expensive; it may be difficult or even impossible.
5. Activities such as backup, security, access control and recovery from error are centralised. They are the responsibility of the repository manager. Tools can focus on their principal function rather than be concerned with these issues.
6. However, different sub-systems may have different requirements for security, recovery and backup policies. The repository model forces the same policy on all sub-systems.
7. The model of sharing is visible through the repository schema. It is straightforward to integrate new tools given that they are compatible with the agreed data model.
8. However, it may be difficult to distribute the repository over a number of machines. Although it is possible to distribute a logically centralised repository, there may be problems with data redundancy and inconsistency.

In the above model, the repository is passive and control is the responsibility of the sub-systems using the repository. An alternative approach has been derived for AI systems that use a 'blackboard' model which triggers sub-systems when particular data becomes available. This is appropriate when the form of the repository data is less well structured. Decisions about which tool to activate can only be made when the data has been analysed. This model is discussed by Nii (1986).

10.1.2 The client–server model

The client–server architectural model is a distributed system model which shows how data and processing are distributed across a range of processors. The major components of this model are:

1. A set of stand-alone servers which offer services to other sub-systems. Examples of servers are print servers which offer printing services, file servers which offer file management services and a compile server which offers programming language compilation services.
2. A set of clients that call on the services offered by servers. These are normally sub-systems in their own right. There may be several instances of a client program executing concurrently.
3. A network which allows the clients to access these services. In principle, this is not really necessary as both the clients and the servers could run on a single machine. In practice, however, this model would not be used in such a situation.

Clients may have to know the names of the available servers and the services that they provide. However, servers need not know either the identity of clients or how many clients there are. Clients access the services provided by a server through remote procedure calls.

An example of a system built around a client–server model is shown in Figure 10.3. This is a multi-user hypertext system to provide a film and photograph library. In this system, there are several servers which manage and display the different types of media. Video frames need to be transmitted quickly and in synchrony but at relatively low resolution. They may be compressed in a store. Still pictures, however, must be sent at a high resolution. The catalogue must be able to deal with

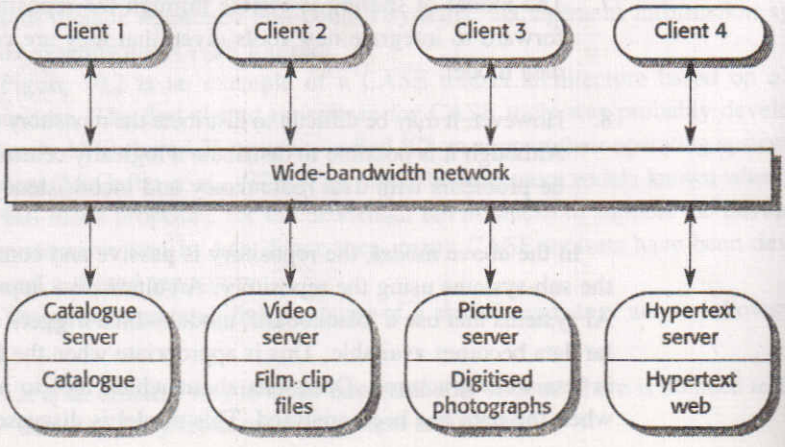


Figure 10.3
The architecture of a film and picture library system

a variety of queries and provide links into the hypertext information systems. The client program is simply an integrated user interface to these services.

The client–server approach can be used to implement a repository-based system where the repository is provided as a system server. Sub-systems accessing the repository are clients. Normally, however, each sub-system manages its own data. Servers and clients exchange data for processing. This can result in performance problems when large amounts of data are exchanged. However, as faster networks are developed, this problem is becoming less significant.

The most important advantage of the client–server model is that it is a distributed architecture. Effective use can be made of networked systems with many distributed processors. It is easy to add a new server and integrate it with the rest of the system or to upgrade servers transparently without affecting other parts of the system. I discuss distributed architectures, including client–server architectures and distributed object architectures, in more detail in Chapter 11.

However, changes to existing clients and servers may be required to gain the full benefits of integrating a new server. There is no shared data model and sub-systems usually organise their data in different ways. This means that specific data models may be established for each server which allow its performance to be optimised. The lack of a shared reference model for data may mean that it is difficult to anticipate problems in integrating data from a new server. Each server must take responsibility for data management activities such as backup and recovery.

10.1.3 The abstract machine model

The abstract machine model of an architecture (sometimes called a layered model) models the interfacing of sub-systems. It organises a system into a series of layers each of which provides a set of services. Each layer defines an *abstract machine* whose machine language (the services provided by the layer) is used to implement the next level of abstract machine. For example, a common way to implement a language is to define an ideal 'language machine' and compile the language into code for this machine. A further translation step then converts this abstract machine code to real machine code.

A well-known example of this approach is the OSI reference model of network protocols (Zimmermann, 1980) which is discussed in section 10.4. Another influential example was proposed by Buxton (1980) who suggested a three-layer model for an Ada Programming Support Environment (APSE). Figure 10.4 has something in common with this and shows how a version management system might be integrated using this abstract machine approach.

The version management system relies on managing versions of objects and provides general configuration management facilities as discussed in Chapter 29. To support these configuration management facilities, it uses an object management system which provides information storage and management services for objects. This system uses a database system to provide basic data storage and services such as transaction management, rollback and recovery, and access control. The

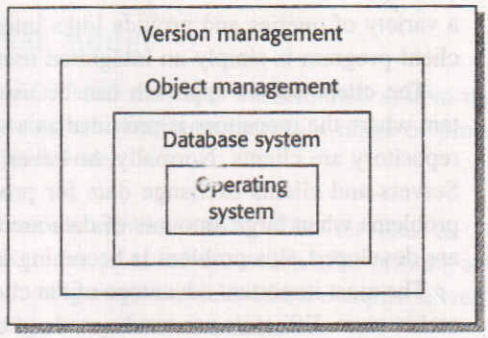


Figure 10.4 Abstract machine model of a version management system

database management uses the underlying operating system facilities and filestore in its implementation.

The layered approach supports the incremental development of systems. As a layer is developed, some of the services provided by that layer may be made available to users. This architecture is also changeable and portable. If its interface is preserved, a layer can be replaced by another layer. When layer interfaces change, only the adjacent layer is affected. As layered systems localise machine dependencies in inner layers, they can be implemented on other computers relatively cheaply. Only the inner, machine-dependent layers need be reimplemented.

A disadvantage of the layered approach is that structuring systems in this way can be difficult. Basic facilities, such as file management, which are required by all abstract machines, may be provided by inner layers. Services required by the user may therefore require access to an abstract machine that is several levels beneath the outermost layer. This subverts the model as an outer layer is no longer simply dependent on its immediate predecessor.

Performance can also be a problem because of the multiple levels of command interpretation which are sometimes required. If there are many layers, some overhead is always associated with layer management. To avoid these problems, applications may have to communicate directly with inner layers rather than use the facilities provided in the abstract machine.

10.2 Control models

The models for structuring a system are concerned with how a system is decomposed into sub-systems. To work as a system, sub-systems must be controlled so that their services are delivered to the right place at the right time. Structural models do not (and should not) include control information. Rather, the architect should organise the sub-systems according to some control model which supplements the

structure model which is used. Control models at the architectural level are concerned with the control flow between sub-systems.

Two general approaches to control can be identified:

1. *Centralised control* One sub-system has overall responsibility for control and starts and stops other sub-systems. It may also devolve control to another sub-system but will expect to have this control responsibility returned to it.
2. *Event-based control* Rather than control information being embedded in a sub-system, each sub-system can respond to externally generated events. These events might come from other sub-systems or from the environment of the system.

Control models supplement structural models. All the above structural models may be realised using centralised or event-based control.

10.2.1 Centralised control

In a centralised control model, one sub-system is designated as the system controller and has responsibility for managing the execution of other sub-systems. Centralised control models fall into two classes depending on whether the controlled sub-systems execute sequentially or in parallel.

1. *The call–return model* This is the familiar top-down subroutine model where control starts at the top of a subroutine hierarchy and, through subroutine calls, passes to lower levels in the tree. The subroutine model is only applicable to sequential systems.
2. *The manager model* This is applicable to concurrent systems. One system component is designated as a system manager and controls the starting, stopping and coordination of other system processes. A process is a sub-system or module which can execute in parallel with other processes. A form of this model may also be applied in sequential systems where a management routine calls particular sub-systems depending on the values of some state variables. This is usually implemented as a case statement.

The call–return model is illustrated in Figure 10.5. The main program can call Routines 1, 2 and 3, Routine 1 can call Routines 1.1 or 1.2, Routine 3 can call Routines 3.1 or 3.2, etc. This is a model of the program dynamics. It is *not* a structural model; there is no need for Routine 1.1, for example, to be part of Routine 1.

This familiar model is embedded in programming languages such as Ada, Pascal and C. Control passes from a higher-level routine in the hierarchy to a lower-level routine. It then returns to the point where the routine was called. The currently executing subroutine has responsibility for control and can either call other routines or return control to its parent. It is poor programming style to return to some other point in the program.

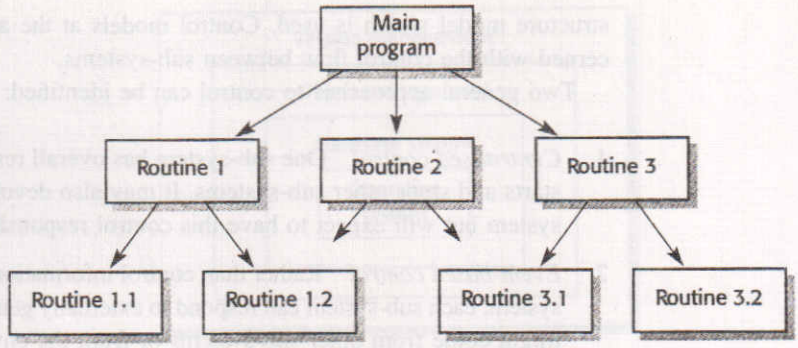


Figure 10.5 The call–return model of control

This call–return model may be used at the module level to control functions or objects. Subroutines in a programming language that are called by other subroutines are naturally functional. However, in many object-oriented systems, operations on objects (methods) are implemented as procedures or functions. For example, when a Java object requests a service from another object, it does so by calling an associated method.

The rigid and restricted nature of this model is both a strength and a weakness. It is a strength because it is relatively simple to analyse control flows and work out how the system will respond to particular inputs. It is a weakness because, as I discuss in Chapter 18, exceptions to normal operation are awkward to handle.

Figure 10.6 is an illustration of a centralised management model of control for a concurrent system. This model is often used in 'soft' real-time systems which do not have very tight time constraints. The central controller manages the execution of a set of processes associated with sensors and actuators. The building monitoring system discussed in Chapter 13 follows this model of control.

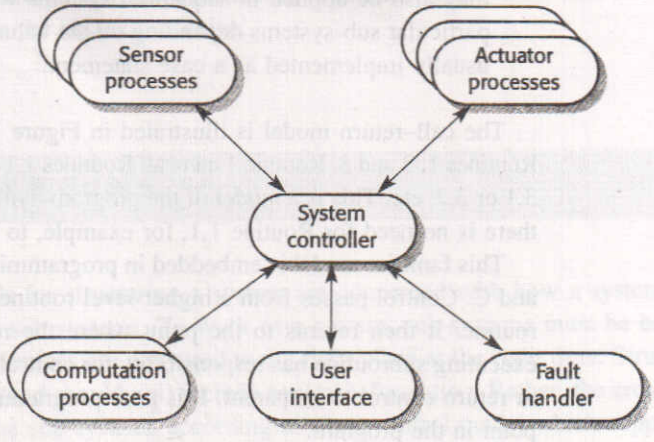


Figure 10.6 A centralised control model for a real-time system

The system controller process decides when processes should be started or stopped depending on system state variables. It checks if other processes have produced information to be processed or to pass information to them for processing. The controller usually loops continuously, polling sensors and other processes for events or state changes. For this reason, this model is sometimes called an event-loop model.

10.2.2 Event-driven systems

In centralised control models, control decisions are usually determined by the values of some system state variables. By contrast, event-driven control models are driven by externally generated events. The term *event* in this context does not just mean a binary signal. It may be a signal which can take a range of values. The distinction between an event and a simple input is that the timing of the event is outside the control of the process which handles that event. A sub-system may need to access state information to handle these events but this state information does not usually determine the flow of control.

There are a variety of different types of event-driven systems which may be developed. These include spreadsheets where changing the value of a cell causes other cells to be modified, rule-based production systems as used in AI where a condition becoming true causes an action to be triggered, and active objects where changing a value of an object's attribute triggers some actions. Garlan *et al.* (1992) discuss these different types of system.

In this section, I discuss two event-driven control models:

1. *Broadcast models* In these models, an event is, in principle, broadcast to all sub-systems. Any sub-system which can handle that event responds to it.
2. *Interrupt-driven models* These are exclusively used in real-time systems where external interrupts are detected by an interrupt handler. They are then passed to some other component for processing.

Broadcast models are effective in integrating sub-systems distributed across different computers on a network. Interrupt-driven models are used in real-time systems with stringent timing requirements.

In a broadcast model (Figure 10.7), sub-systems register an interest in specific events. When these events occur, control is transferred to the sub-system which can handle the event. The distinction between this model and the centralised model shown in Figure 10.6 is that the control policy is not embedded in the event and message handler. Sub-systems decide which events they require and the event and message handler ensures that these events are sent to them.

All events could be broadcast to all sub-systems but this imposes a great deal of processing overhead. More often, the event and message handler maintains a register of sub-systems and the events of interest to them. Sub-systems generate

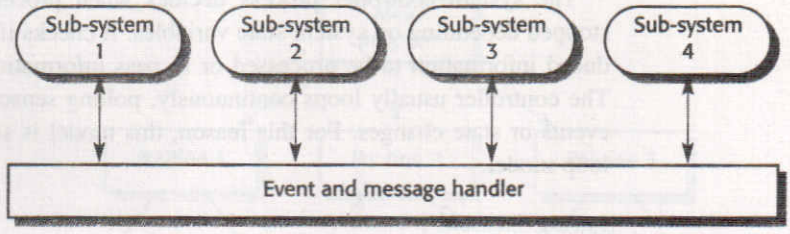


Figure 10.7
A control model based on selective broadcasting.

events indicating, perhaps, that some data is available for processing. The event handler detects the events, consults the event register and passes the event to those sub-systems which have declared an interest.

The event handler also usually supports point-to-point communication. A sub-system can explicitly send a message to another sub-system. There have been a number of variations of this model such as the Field environment (Reiss, 1990) and Hewlett-Packard's Softbench (Fromme and Walker, 1993). Both of these have been used to control tool interactions in software engineering environments. Object request brokers, discussed in Chapter 11, also support this model of control for distributed object communications.

The advantage of this broadcast approach is that evolution is relatively simple. A new sub-system to handle particular classes of events can be integrated by registering its events with the event handler. Any sub-system can activate any other sub-system without knowing its name or location. The sub-systems can be implemented on distributed machines. This distribution is transparent to other sub-systems.

The disadvantage of this model is that sub-systems don't know if or when events will be handled. When a sub-system generates an event it does not know which other sub-systems have registered an interest in that event. It is quite possible for different sub-systems to register for the same events. This may cause conflicts when the results of handling the event are made available.

Real-time systems that require externally generated events to be handled very quickly must be event-driven. For example, if a real-time system is used to control the safety systems in a car, it must detect a possible crash and, perhaps, inflate an airbag before the driver's head hits the steering wheel. To provide this rapid response to events, you have to use interrupt-driven control.

An interrupt-driven control model is illustrated in Figure 10.8. There are a known number of interrupt types with a handler defined for each type. Each type of interrupt is associated with the memory location where its handler's address is stored. When an interrupt of a particular type is received, a hardware switch causes control to be transferred immediately to its handler. This interrupt handler may then start or stop other processes in response to the event signalled by the interrupt.

This model should only be used in hard real-time systems where immediate response to some event is necessary. It may be combined with the centralised management model. The central manager handles the normal running of the system with interrupt-based control for emergencies.

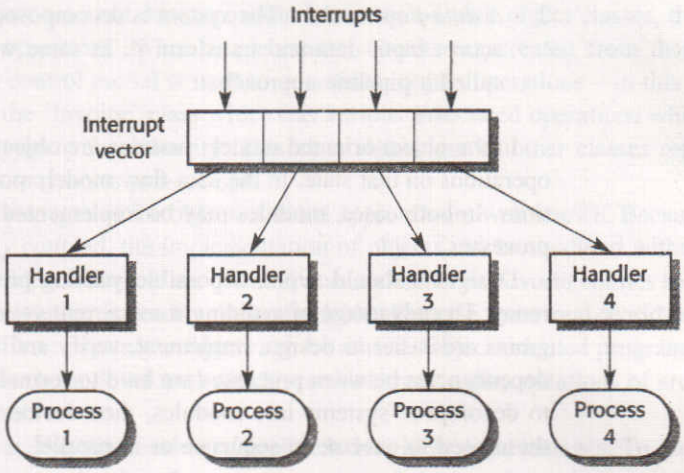


Figure 10.8 An interrupt-driven control model

The advantage of this approach to control is that it allows very fast responses to events to be implemented. Its disadvantages are that it is complex to program and difficult to validate. It may be impossible to replicate patterns of interrupt timing during the system testing process. It can be difficult to change systems developed using this model if the number of interrupts is limited by the hardware. Once this limit is reached, no other types of event can be handled. This limitation can sometimes be circumvented by mapping several types of event onto a single interrupt, then leaving the handler to work out which event has occurred. However, this may be an impractical approach if a very fast response to the interrupt is required.

10.3 Modular decomposition

After a structural architecture has been designed, the next stage of the architectural design process is the decomposition of sub-systems into modules. There is not a rigid distinction between system decomposition and modular decomposition. The models discussed in section 10.1 could be applied at this level. However, the components in modules are usually smaller than sub-systems and this allows alternative decomposition models to be used.

I consider two models which may be used when decomposing a sub-system into modules:

1. *An object-oriented model* The system is decomposed into a set of communicating objects.

2. *A data-flow model* The system is decomposed into functional modules which accept input data and transform it, in some way, to output data. This is also called a pipeline approach.

In the object-oriented model, modules are objects with private state and defined operations on that state. In the data-flow model, modules are functional transformations. In both cases, modules may be implemented as sequential components or as processes.

Designers should avoid, if possible, making premature decisions about concurrency. The advantage of avoiding a concurrent system design is that sequential programs are easier to design, implement, verify and test than parallel systems. Time dependencies between processes are hard to formalise, control and verify. It is best to decompose systems into modules, then decide during implementation whether these need to execute in sequence or in parallel.

10.3.1 Object models

An object-oriented model of a system architecture structures the system into a set of loosely coupled objects with well-defined interfaces. Objects call on the services offered by other objects. I have already introduced object models in Chapter 7 and I discuss them in more detail in Chapter 12.

Figure 10.9 is an example of an object-oriented architectural model of an invoice processing system. This system can issue invoices to customers, receive payments, issue receipts for these payments and reminders for unpaid invoices. I use the UML notation introduced in Chapter 7 where object classes have names and a set of associated attributes. Operations, if any, are defined in the lower part of the rounded rectangle representing the object. Dashed arrows indicate that an object uses the attributes or services provided by another object.

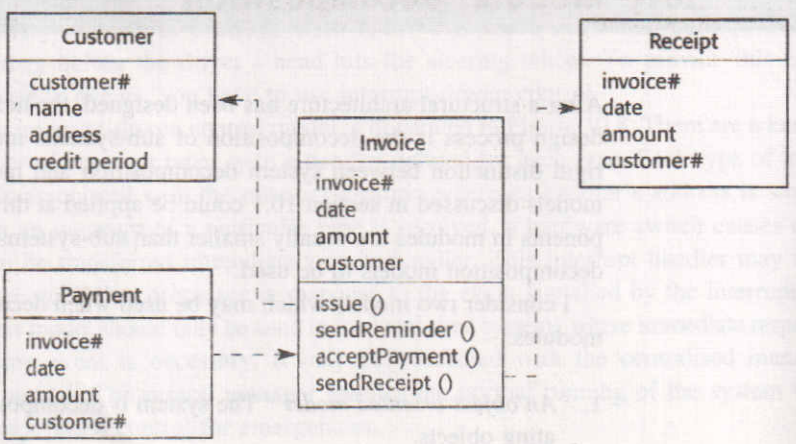


Figure 10.9 An object model of an invoice processing system

An object-oriented decomposition is concerned with object classes, their attributes and operations. When implemented, objects are created from these classes and some control model is used to coordinate object operations – in this particular example, the 'Invoice' class which has various associated operations which implement the system functionality. This class makes use of other classes representing customers, payments and receipts.

The advantages of the object-oriented approach are well known. Because objects are loosely coupled, the implementation of objects can be modified without affecting other objects. Objects are often representations of real-world entities so the structure of the system is readily understandable. Because these real-world entities are used in different systems, objects can be reused. Object-oriented programming languages have been developed which provide direct implementations of architectural components.

However, the object-oriented approach does have disadvantages. To use services, objects must explicitly reference the name and the interface of other objects. If an interface change is required to satisfy proposed system changes, the effect of that change on all users of the changed object must be evaluated. While objects may map cleanly to small-scale real-world entities, more complex entities are sometimes difficult to represent as objects.

10.3.2 Data-flow models

In a data-flow model, functional transformations process their inputs and produce outputs. Data flows from one to another and is transformed as it moves through the sequence. Each processing step is implemented as a transform. Input data flows through these transforms until converted to output. The transformations may execute sequentially or in parallel. The data can be processed by each transform item by item or in a single batch.

When the transformations are represented as separate processes, this model is sometimes called the pipe and filter model after the terminology used in the Unix system. The Unix system provides pipes which act as data conduits and a set of commands which are functional transformations. Systems which conform to this model can be implemented by combining Unix commands using pipes and the control facilities of the Unix shell. The term 'filter' is used because a transformation 'filters out' the data it can process from its input data stream.

Variants of this data-flow model have been in use since computers were first used for automatic data processing. When transformations are sequential with data processed in batches, this architectural model is a batch sequential model. This is a common architecture for some classes of data processing systems such as billing systems which generate large numbers of output reports that are derived from simple computations on a large number of input records.

An example of this type of system architecture is shown in Figure 10.10. An organisation has issued invoices to customers. Once a week, payments which have

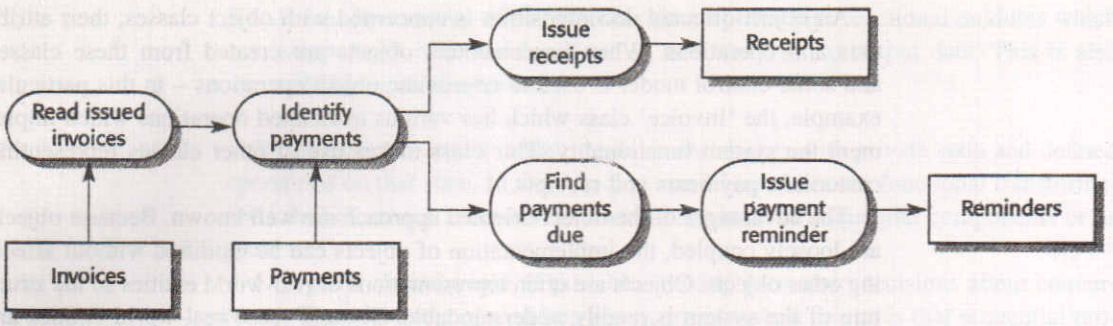


Figure 10.10
A data-flow model of an invoice processing system

been made are reconciled with the invoices. For those invoices that have been paid, a receipt is issued. For those invoices that have not been paid within the allowed payment time, a reminder is issued.

This is a model of only part of the invoice processing system; alternative transformations would be used for the issue of invoices. Notice the difference between this and its object-oriented equivalent discussed in the previous section. The object model is more abstract as it does not include information about the sequence of operations.

The advantages of this architecture are:

1. It supports the reuse of transformations.
2. It is intuitive in that many people think of their work in terms of input and output processing.
3. Evolving the system by adding new transformations is usually straightforward.
4. It is simple to implement either as a concurrent or a sequential system.

The principal disadvantage of the model stems from the need for a common format for data transfer which can be recognised by all transformations. Each transformation must either agree with its communicating transformations on the format of the data which will be processed, or a standard format for all data communicated must be imposed. The latter approach is the only feasible approach when transformations are stand-alone and reusable. In Unix, the standard format is simply a character sequence. Each transformation must parse its input and unparse its output to the agreed form. This increases system overhead and may mean that it is impossible to integrate transformations that use incompatible data formats.

Interactive systems are difficult to write using the data-flow model because of the need for a stream of data to be processed. While simple textual input and output can be modelled in this way, graphical user interfaces have more complex I/O formats and control which is based on events such as mouse clicks or menu selections. It is difficult to translate this into a form compatible with the data-flow model.

10.4 Domain-specific architectures

The above architectural models are general models. They can be applied to many different classes of application. As well as these general models, architectural models which are specific to a particular application domain may also be used. Although instances of these systems differ in detail, the common architectural structure can be reused when developing new systems. These architectural models are called *domain-specific architectures*.

There are two types of domain-specific architectural model:

1. *Generic models* which are abstractions from a number of real systems. They encapsulate the principal characteristics of these systems. For example, in real-time systems, there might be generic architectural models of different system types such as data collection systems, monitoring systems, etc.
2. *Reference models* which are more abstract and describe a larger class of systems. They are a way of informing designers about the general structure of that class of system. For example, Rockwell and Gera (1993) have proposed a reference model for software factories.

There is not, of course, a rigid distinction between these different types of model. Generic models can also sometimes serve as reference models. I make a distinction between them here because generic models may be reused directly in a design. Reference models are normally used to communicate domain concepts and compare possible architectures.

This reflects the derivation of these models. Generic models are usually derived 'bottom-up' from existing systems whereas reference models are derived 'top-down'. They are abstract system representations. Reference models do not necessarily reflect the actual architecture of existing systems in the domain.

10.4.1 Generic models

Perhaps the best known example of a generic architectural model is a compiler model. Thousands of compilers have been written. It is now generally agreed that compilers should include the following modules:

1. A lexical analyser which takes input language tokens and converts them to some internal form.
2. A symbol table, built by the lexical analyser, which holds information about the names and types used in the program.
3. A syntax analyser which checks the syntax of the language being compiled. It uses a defined grammar of the language and builds a syntax tree.

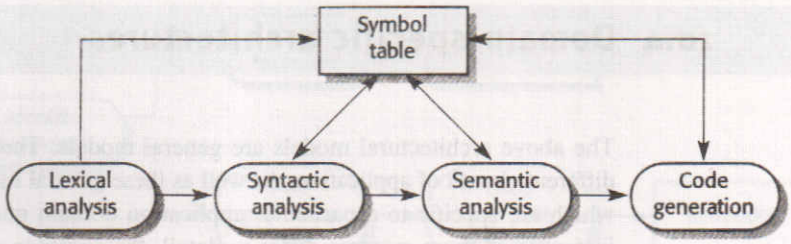


Figure 10.11
A data-flow model of a compiler

4. A syntax tree which is an internal structure representing the program being compiled.
5. A semantic analyser which uses information from the syntax tree and the symbol table to check the semantic correctness of the input program.
6. A code generator which 'walks' the syntax tree and generates machine code.

Other components might also be included which transform the syntax tree to improve efficiency and remove redundancy from the generated machine code.

The components which make up a compiler can be organised according to different architectural models. As Garlan and Shaw point out (1993), compilers can be implemented using a composite model. A data-flow architecture may be used, with the symbol table acting as a repository for shared data. The phases of lexical, syntactic and semantic analysis are organised sequentially as shown in Figure 10.11.

This model is still widely used. It is effective in batch environments where programs are compiled and executed without user interaction. However, it is less effective when the compiler is to be integrated with other language processing tools such as a structured editing system, an interactive debugger, a program prettyprinter, etc. The generic system components can then be organised in a repository-based model as shown in Figure 10.12.

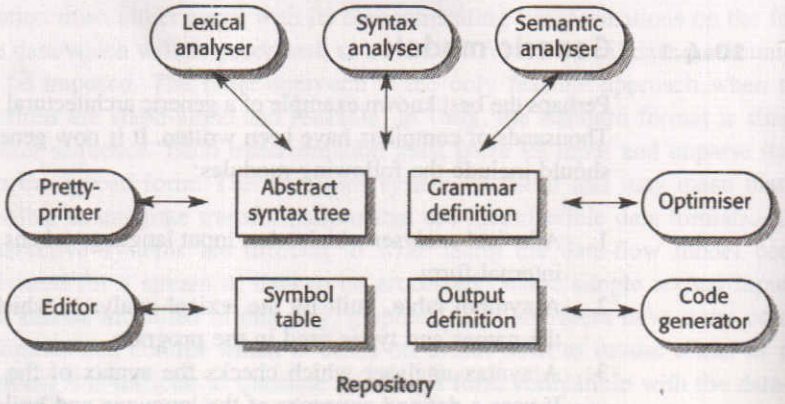


Figure 10.12
The repository model of a language processing system

In this model of a compiler, the symbol table and syntax tree act as a central information repository. Tools or tool fragments communicate through it. Other information such as the grammar definition and the definition of the output format for the program have been taken out of the tools and into the repository.

There are, in practice, a very large number of domain-specific architectural models. Some further examples of generic process architectures for real-time systems are illustrated in Chapter 13.

However, relatively few generic, domain-specific models are publicly available. Organisations which develop these models see them as valuable intellectual property which they need for future system development. They often represent the architecture of a product line where related products are implemented using the same basic architecture. For example, printer drivers for printers with different capabilities (speed, paper options, etc.) may all use the same fundamental architecture. I discuss product-line architectures in Chapter 14.

10.4.2 Reference architectures

Generic architectural models reflect the architecture of existing systems. In contrast, reference models are usually derived from a study of the application domain. They represent an idealised architecture which includes all the features that systems might incorporate.

Reference architectures may be used as a basis for system implementation. This was the intention behind the OSI reference model (Zimmermann, 1980) for open systems interconnection. The model was intended as a standard. If a system conformed to the model, it should be able to communicate with other conformant systems. Thus, a stock control system in a supermarket which followed the OSI model could exchange data directly with the supplier's ordering system.

However, reference models should not normally be considered as a route to implementation. Rather, their principal function is to serve as a means of comparing different systems in a domain. A reference model provides a vocabulary for comparison. It acts as a standard, against which systems can be evaluated.

The OSI model is a seven-layer model for open systems interconnection. The model is illustrated in Figure 10.13. The exact functions of the different layers are not important here. In essence, the lower layers are concerned with physical interconnection, the middle layers with data transfer and the upper layers with the transfer of semantically meaningful application information such as standardised documents, etc.

The designers of the OSI model had the very practical objective of defining a standard so that conformant systems could communicate with each other. Each layer should only depend on the layer beneath it. As technology developed, a layer could be transparently reimplemented without affecting the systems using other layers.

In practice, however, the problems of the layered approach to architectural modelling have compromised this objective. Because of the vast differences between networks, simple interconnection may be impossible. Although the functional

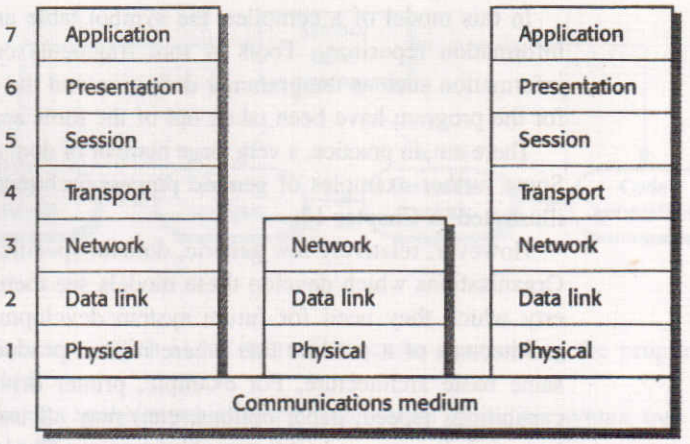


Figure 10.13
The OSI reference model architecture

characteristics of each layer are well defined, the non-functional characteristics are not defined. System developers have to implement their own higher-level facilities and skip layers in the model. Alternatively, they have to design non-standard features to improve system performance.

Consequently, the transparent replacement of a layer in the model is hardly ever possible. However, this does not negate the usefulness of the model as it provides a basis for the abstract structuring and the systematic implementation of communications between systems.

Other reference models which have been proposed include a model for CASE environments (ECMA, 1991) and a model for software factories (Rockwell and Gera, 1993). Some design patterns (Gamma *et al.*, 1995) may also be considered as reference architectures. These are discussed in Chapter 14.

KEY POINTS

- The software architecture is the fundamental framework for structuring the system. Different architectural models such as a structural model, a control model and a decomposition model may be developed during architectural design.
- Large systems rarely conform to a single architectural model. They are heterogeneous and incorporate different models at different levels of abstraction.
- System decomposition models include repository models, client–server models and abstract machine models. Repository models share data through a common store. Client–server models usually distribute data. Abstract machine models are layered, with each layer implemented using the facilities provided by its foundation layer.

- Control models include centralised control and event models. In centralised models, control decisions are made depending on the system state; in event models external events control the system.
- Modular decomposition models include data-flow and object models. Data-flow models are functional whereas object models are based on loosely coupled entities that maintain their own state and operations.
- Domain-specific architectural models are abstractions over an application domain. Domain-specific models may be generic models which are constructed bottom-up from existing systems or reference models which are idealised, abstract models of the domain.

FURTHER READING

Software architecture is a hot topic and there are a large number of books available. Books that I have found useful are:

Software Architecture in Practice. This is a practical discussion of software architectures that does not oversell the approach and that provides a clear business rationale why architectures are important. (L. Bass, P. Clements and R. Kazman, 1998, Addison-Wesley.)

Systems Engineering: Coping with Complexity. Chapter 4 of this book on systems engineering discusses the architectural design of systems rather than simply software. This is a refreshingly different perspective. (R. Stevens, P. Brook, K. Jackson and S. Arnold, 1998, Prentice-Hall.)

Software Architecture: Perspectives on an Emerging Discipline. This was the first book on software architecture and has a good discussion on different architectural styles. (M. Shaw and D. Garlan, 1996, Prentice-Hall.)

EXERCISES

10.1 Explain why it may be necessary to design the system architecture before the specifications are written.

10.2 Construct a table showing the advantages and disadvantages of the different structural models discussed in this chapter.

10.3 Giving reasons for your answer, suggest an appropriate structural model for the following systems:

- an automated ticket issuing system used by passengers at a railway station;
- a computer-controlled video conferencing system which allows video, audio and computer data to be visible to several participants at the same time;
- a robot floor cleaner which is intended to clean relatively clear spaces such as corridors. The cleaner must be able to sense walls and other obstructions.

10.4 Design an architecture for the above systems based on your choice of model. Make reasonable assumptions about the system requirements.

10.5 Explain why a call–return model of control is not usually suitable for real-time systems which control some process.

10.6 Giving reasons for your answer, suggest an appropriate control model for the following systems:

- a batch processing system which takes information about hours worked and pay rates and prints salary slips and bank credit transfer information;
- a set of software tools which are produced by different vendors but which must work together;
- a television controller which responds to signals from a remote control unit.

10.7 Discuss their advantages and disadvantages as far as distributability is concerned of the data-flow model and the object model. Assume that both single machine and distributed versions of an application are required.

10.8 You are given two integrated CASE toolsets and are asked to compare them. Explain how you could use a reference model for CASE (Brown *et al.*, 1992) to make this comparison.

10.9 Should there be a separate profession of 'software architect' whose role is to work independently with a customer to design a software architecture? This would then be implemented by some software company. What might be the difficulties of establishing such a profession?