7 System models

Objectives

The aim of this chapter is to introduce a number of different types of system model which may be developed during the requirements engineering process. When you have read the chapter, you will:

- understand why it is important to model the context of a system;
- understand the concepts of behavioural modelling, data modelling and object modelling;
- have been introduced to some of the notations defined in the Unified Modeling Language (UML) and how these notations may be used to develop different types of system model;
- understand how CASE workbenches support system modelling.

Contents

User requirements must be written in natural language because they have to be understood by people who are not technical experts. However, more detailed system requirements may be expressed in a more technical way. One widely used technique is to document the system specification as a set of system models. These models are graphical representations that describe the problem to be solved and the system that is to be developed. Because of the graphical representations used, models are often more understandable than detailed natural language descriptions of the system requirements. They are also an important bridge between the analysis and design processes.

Models may be used in the analysis process to develop an understanding of the existing system that is to be replaced or improved or to specify the required system. They can be used to represent the system from different perspectives:

1. an external perspective where the context or environment of the system is modelled;
2. a behavioural perspective where the behaviour of the system is modelled;
3. a structural perspective where the architecture of the system or the structure of the data processed by the system is modelled.

I cover these three perspectives in this chapter and also discuss object modelling which combines, to some extent, behavioural and structural modelling.

Structured methods such as structured systems analysis (DeMarco, 1978) and object-oriented analysis (Rumbaugh *et al.*, 1991; Booch, 1994) provide a framework for detailed system modelling as part of requirements elicitation and analysis. Most structured methods have their own preferred set of system models. They usually define a process which may be used to derive these models and a set of rules and guidelines which apply to the models. Standard documentation is produced for the system. CASE tools (discussed in section 7.5) are usually available for method support. These tools include model editors, automate system documentation and provide some model checking capabilities.

However, structured analysis methods suffer from a number of weaknesses:

1. They do not provide effective support for understanding or modelling non-functional system requirements.
2. They are indiscriminate in that they do not usually include guidelines to help users decide whether or not a method is appropriate for a particular problem. Nor do they normally include advice on how they may be adapted for use in a particular environment.
3. They often produce too much documentation. The essence of the system requirements may be hidden by the mass of detail which is included.
4. The models that are produced are very detailed and users often find them difficult to understand. These users therefore cannot really check the realism of these models.

In practice, requirements engineers don't have to restrict themselves to the models proposed in any particular method. For example, object-oriented methods

do not usually suggest that data-flow models should be developed. However, in my experience, such models are sometimes useful as part of an object-oriented analysis process. They often reflect the end-user's understanding of the system. They may also contribute directly to object identification (the data which flows) and the identification of operations on these objects.

The most important aspect of a system model is that it leaves out detail. A system model is an abstraction of the system being studied rather than an alternative representation of that system. Ideally, a *representation* of a system should maintain all the information about the entity being represented. An *abstraction* deliberately simplifies and picks out the most salient characteristics. For example, in the unlikely event of this book being summarised in the *Reader's Digest*, the presentation there would be an abstraction of the key points. If it was translated from English into Italian, this would be an alternative representation. The translator's intention would be to maintain all the information as it is presented in English.

Different types of system model are based on different approaches to abstraction. A data-flow model (for example) concentrates on the flow of data and the functional transformations on that data. It leaves out details of the data structures. By contrast, an entity-relation model is intended to document the system data and its relationships without concern for the functions in the system.

Examples of the different types of system model which might be produced during the analysis process are:

1. *A data-processing model* Data-flow diagrams show how data is processed at different stages in the system.
2. *A composition model* Entity-relation diagrams show how entities in the system are composed of other entities.
3. *An architectural model* Architectural models show the principal sub-systems which make up a system.
4. *A classification model* Object class/inheritance diagrams show how entities have common characteristics.
5. *A stimulus-response model* State transition diagrams show how the system reacts to internal and external events.

All these types of model are covered in this chapter. Wherever possible, I use notations from the Unified Modeling Language (UML) which is emerging as a standard modelling language, particularly for object-oriented modelling (Booch *et al.*, 1999; Rumbaugh *et al.*, 1999a). Where UML does not include appropriate notations, I use simple intuitive notations for model description.

7.1 Context models

At an early stage in the requirements elicitation and analysis process you should decide on the boundaries of the system. This involves working with system stake-

holders to distinguish what is the system and what is the system's environment. You should make these decisions early in the process to limit the system costs and the time needed for analysis.

In some cases, the boundary between a system and its environment is relatively clear. For example, where an automated system is replacing an existing manual or computerised system, the environment of the new system is usually the same as the existing system's environment. In other cases, there is more flexibility. You decide on the boundary between the system and its environment during the requirements engineering process.

For example, the environment of a CASE toolset may include an existing database whose services are used by the system or the toolset may define its own internal database. Given that a database already exists, the positioning of the boundary between these systems may be a difficult technical and managerial problem. It is only possible to make a decision about what is and what is not part of the system after you have done some analysis.

The definition of a system context is not a value-free judgement. Social and organisational concerns may mean that the position of a system boundary may be determined by non-technical factors. For example, a system boundary may be positioned so that the analysis process can all be carried out on one site; it may be chosen so that a particularly difficult manager need not be consulted; it may be positioned so that the system cost is increased and the system development division must therefore expand to design and implement the system.

Once some decisions on the boundaries of the system have been made, part of the analysis activity is the definition of that context and the dependencies that a system has on its environment. Normally, producing a simple architectural model is the first step in this activity.

This is illustrated in Figure 7.1. This is an architectural model which illustrates the structure of the information system that includes a bank auto-teller network. High-level architectural models are usually expressed as simple block diagrams where each sub-system is represented by a named rectangle and lines indicate that there are some associations between sub-systems.

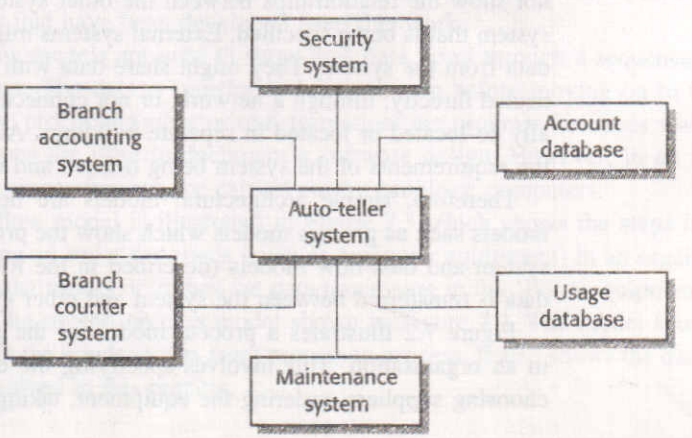


Figure 7.1
The context of the ATM system

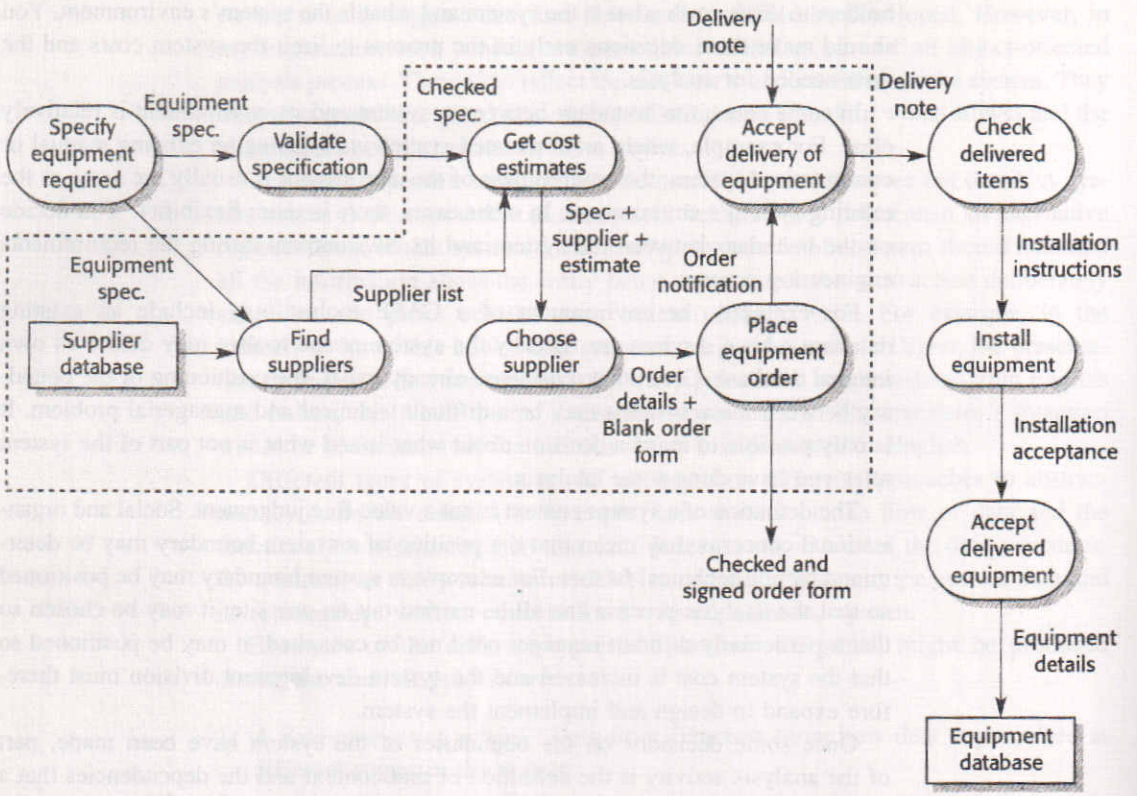


Figure 7.2 Process model of equipment procurement

From Figure 7.1 we see that each auto-teller machine is connected to an account database, a local branch accounting system, a security system and a system to support machine maintenance. The system is also connected to a usage database which monitors how the network of ATMs is used and to a local branch counter system. This counter system provides services such as backup and printing. These, therefore, need not be included in the auto-teller itself.

Architectural models describe the environment of a system. However, they do not show the relationships between the other systems in the environment and the system that is being specified. External systems might produce data for or consume data from the system. They might share data with the system, they might be connected directly, through a network, or not connected at all. They might be physically co-located or located in separate buildings. All of these relations might affect the requirements of the system being defined and must be taken into account.

Therefore, simple architectural models are normally supplemented by other models such as process models which show the process activities supported by the system and data-flow models (described in the following section) that show how data is transferred between the system and other systems in its environment.

Figure 7.2 illustrates a process model for the process of procuring equipment in an organisation. This involves specifying the equipment required, finding and choosing suppliers, ordering the equipment, taking delivery of the equipment and

testing it after delivery. When specifying computer support for this process you have to decide which of these activities will actually be supported. The other activities are outside the boundary of the system. In Figure 7.2, the dotted line encloses the activities that are within the system boundary.

7.2 Behavioural models

Behavioural models are used to describe the overall behaviour of the system. I discuss two types of behavioural model, namely data-flow models which model the data processing in the system and state machine models which model how the system reacts to events. These models may be used separately or together, depending on the type of system which is being developed.

Most business systems are primarily driven by data. They are controlled by the data inputs to the system with relatively little external event processing. A data-flow model may be all that is needed to represent the behaviour of these systems. By contrast, real-time systems are often event-driven with minimal data processing. A state machine model (discussed in section 7.2.2) is the most effective way to represent their behaviour. Other classes of system may be both data- and event-driven so both types of model should be developed.

7.2.1 Data-flow models

Data-flow models are an intuitive way of showing how data is processed by a system. At the analysis level, they should be used to model the way in which data is processed in the existing system. The notation used in these models represents functional processing, data stores and data movements between functions. The use of data-flow models for analysis became widespread after the publication of DeMarco's book (1978) on structured systems analysis. They are an intrinsic part of methods that have been developed from this work.

Data-flow models are used to show how data flows through a sequence of processing steps. The data is transformed at each step before moving on to the next stage. These processing steps or transformations are program functions when data-flow diagrams are used to document a software design. However, in an analysis model, the processing may be carried out by people or computers.

A data-flow model is illustrated in Figure 7.3 which shows the steps involved in processing an order for goods (such as computer equipment) in an organisation. This particular model describes the data processing in the 'Place equipment order' activity in the overall process model shown in Figure 7.2. The model shows how the order for the goods moves from process to process. It also shows the data stores that are involved in this process.

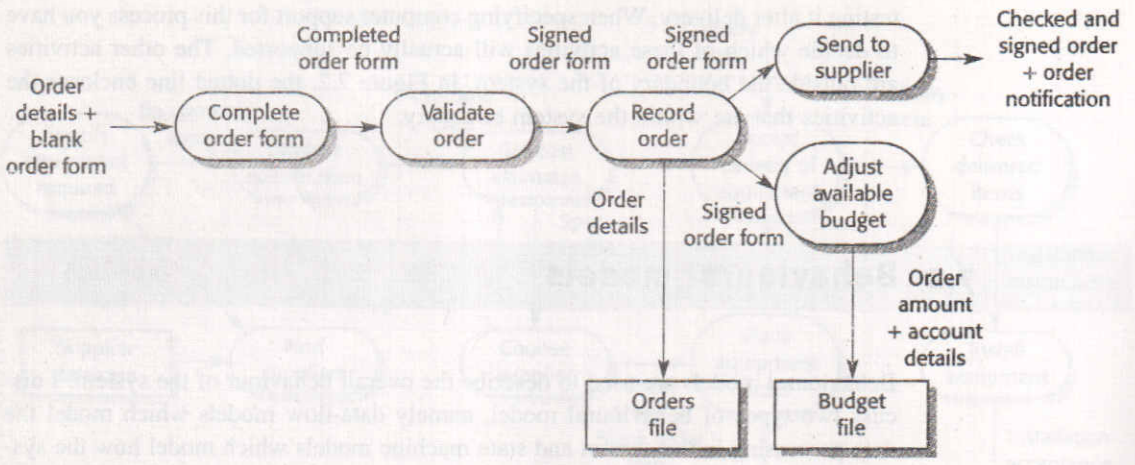


Figure 7.3 Data-flow diagram of order processing

In the notation that I use to describe data-flow diagrams in Figure 7.3, rounded rectangles represent processing steps, arrows annotated with the data name represent flows and rectangles represent data stores or data sources.

Data-flow models are valuable because tracking and documenting how the data associated with a particular process moves through the system helps analysts understand what is going on. Data-flow diagrams have the advantage that, unlike some other modelling notations, they are simple and intuitive. It is often possible to explain them to potential system users who can therefore participate in validating the analysis.

In principle, the development of models such as data-flow models should be a 'top-down' process. In this example, this would imply that the overall procurement process should be analysed first. The analysis of sub-processes such as ordering should then be carried out. In practice, analysis is never like that. Information about the system is normally acquired about several different levels at the same time. Lower-level models may be developed first, then abstracted to create a more general model.

Data-flow models show a functional perspective where each transformation represents a single function. As well as describing processing within a system, it is sometimes useful to use data-flow descriptions to illustrate the context of the system. The data-flow model can show how different systems and sub-systems exchange information. These sub-systems need not be single functions. For example, one sub-system might be a storage server with a fairly complex interface. Figure 7.4 shows an example of a data-flow diagram used in this way. In this example, the rounded rectangles represent sub-systems.

7.2.2 State machine models

State machine models are used to model the behaviour of a system in response to internal or external events. The state machine model shows system states and events which cause transitions from one state to another. It does not show the flow of data within the system. This type of model is particularly useful for modelling real-time systems because these systems are often driven by stimuli from the system's

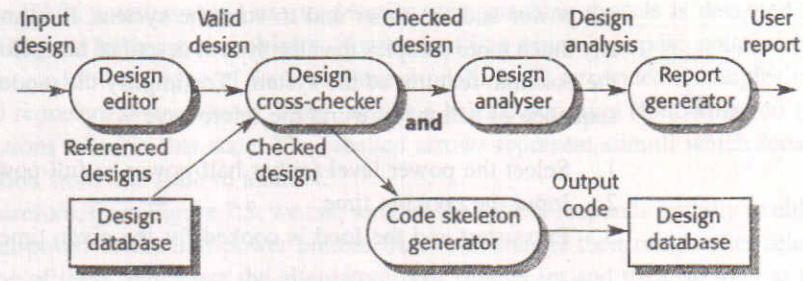


Figure 7.4 Data-flow diagram of a CASE toolset

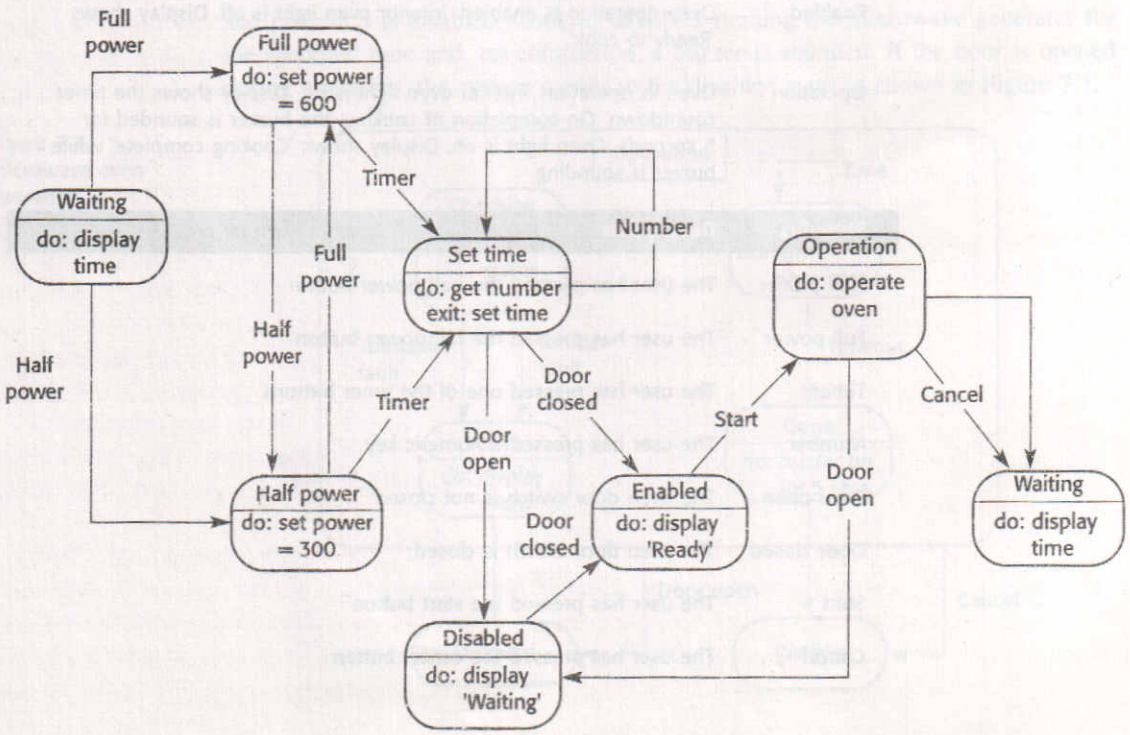
environment. For example, the real-time alarm system discussed in Chapter 13 responds to stimuli from movement sensors, door opening sensors, etc.

State machine models are an integral part of real-time design methods such as that proposed by Ward and Mellor (1985) and Harel (1987, 1988). Harel's method uses a notation called *Statecharts* and these were the basis for the state machine modelling notation in the UML.

A state machine model of a system assumes that, at any time, the system is in one of a number of possible states. When a stimulus is received, this may trigger a transition to a different state. For example, a system controlling a valve may move from a state 'Valve open' to a state 'Valve closed' when an operator command (the stimulus) is received.

Figure 7.5 State machine model of a simple microwave oven

This approach to system modelling is illustrated in Figure 7.5. This diagram shows a state machine model of a simple microwave oven equipped with buttons to set



the power and the timer and to start the system. Real microwave ovens are actually much more complex than the system described here. However, this model includes the essential features of the system. To simplify the model, I have assumed that the sequence of actions in using the microwave is:

1. Select the power level (either half-power or full-power).
2. Input the cooking time.
3. Press start and the food is cooked for the given time.

For safety reasons, the oven should not operate when the door is open and, on completion of cooking, a buzzer is sounded. The oven has a very simple alphanumeric display which is used to display various alerts and warning messages.

Figure 7.6 Microwave oven state and stimulus description

State	Description
Waiting	The oven is waiting for input. The display shows the current time.
Half power	The oven power is set to 300 watts. The display shows 'Half power'.
Full power	The oven power is set to 600 watts. The display shows 'Full power'.
Set time	The cooking time is set to the user's input value. The display shows the cooking time selected and is updated as the time is set.
Disabled	Oven operation is disabled for safety. Interior oven light is on. Display shows 'Not ready'.
Enabled	Oven operation is enabled. Interior oven light is off. Display shows 'Ready to cook'.
Operation	Oven in operation. Interior oven light is on. Display shows the timer countdown. On completion of cooking, the buzzer is sounded for 5 seconds. Oven light is on. Display shows 'Cooking complete' while buzzer is sounding.

Stimulus	Description
Half power	The user has pressed the half power button
Full power	The user has pressed the full power button
Timer	The user has pressed one of the timer buttons
Number	The user has pressed a numeric key
Door open	The oven door switch is not closed
Door closed	The oven door switch is closed
Start	The user has pressed the start button
Cancel	The user has pressed the cancel button

The UML notation that I use to describe state machine models is designed for modelling the behaviour of objects. However, it is a general-purpose notation that can be used for any type of state machine modelling. The rounded rectangles in a model represent system states. They include a brief description (following 'do') of the actions taken in that state. The labelled arrows represent stimuli which force a transition from one state to another.

Therefore, from Figure 7.5, we can see that the system responds initially to either the full-power or the half-power button. Users can change their minds after selecting one of these and select the alternative. The time is set and then, so long as the door is closed, the start button is enabled. Pushing this button starts the oven operation and cooking takes place for the specified time.

The UML notation lets you indicate the activity which takes place in a state. However, in a detailed system specification you have to provide more detail about both the stimuli and the system states (Figure 7.6). This information may be maintained in a data dictionary as discussed later in this section.

The problem with the state machine approach is that the number of possible states increases rapidly. For large system models, therefore, some structuring of these state models is necessary. One way to do this is by using the notion of a superstate which encapsulates a number of separate states. This superstate looks like a single state on a high-level model but is then expanded in more detail on a separate diagram. To illustrate this concept, consider the Operation state in Figure 7.5. This is a superstate which can be expanded as illustrated in Figure 7.7.

The Operation state includes a number of sub-states. It shows that operation starts with a status check and if any problems are discovered then an alarm is indicated and operation is disabled. Cooking involves running the microwave generator for the specified time and, on completion, a buzzer is sounded. If the door is opened during operation, the system moves to the disabled state as shown in Figure 7.5.

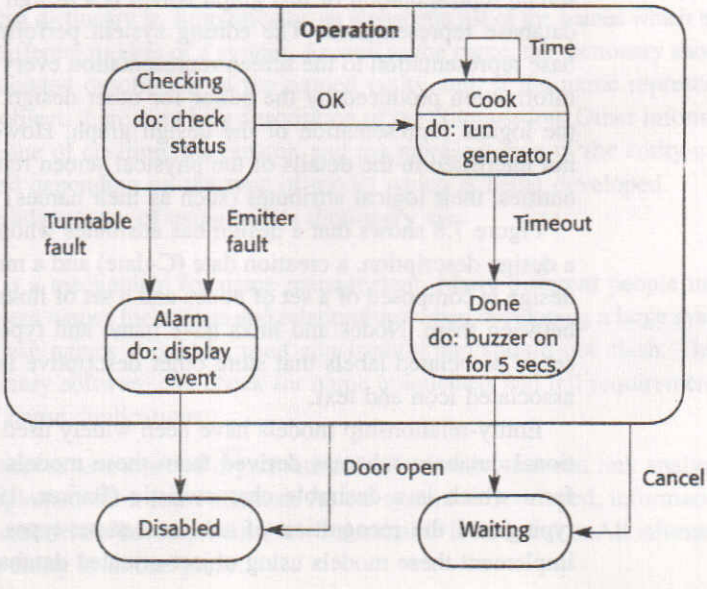


Figure 7.7 Microwave oven operation

7.3 Data models

Most large software systems make use of a large database of information. In some cases, this database exists independently of the software system. In others, it is created for the system being developed. An important part of systems modelling is to define the logical form of the data processed by the system.

The most widely used data modelling technique is entity–relation–attribute modelling (ERA modelling) which shows the data entities, their associated attributes and the relations between these entities. This approach to modelling was first proposed in the mid-1970s by Chen (1976) with several variants developed since then (Codd, 1979; Hammer and McLeod, 1981; Hull and King, 1987). However, all of these have the same basic form.

The UML does not include a specific notation for this type of data modelling as it assumes an object-oriented development process and models data using objects and their relationships. However, you can think of entities as simplified object classes (they have no operations) and you can use UML class models with named associations between the classes as data models. Although the resultant data models are not 'good' UML, this is outweighed by the convenience of using a standard notation.

Data models are often used in conjunction with data-flow models to describe the structure of the information which is being processed. I have illustrated this by including a data model of a software design (Figure 7.8). Such a design could be processed by the various components of the CASE toolset shown in Figure 7.4.

Designs are directed graphs. They consist of a set of nodes of different types connected by links representing the relationships between design nodes. There is a screen representation of this graph which is a design diagram and a corresponding database representation. The editing system performs a mapping from the database representation to the screen representation every time it draws a diagram. The information produced by the editor for other design analysis tools should include the logical representation of the design graph. However, these analysis tools are not interested in the details of the physical screen representation. They process the entities, their logical attributes (such as their names) and their relationships.

Figure 7.8 shows that a design has attributes whose values are the design name, a design description, a creation date (C-date) and a modification date (M-date). The design is composed of a set of nodes and a set of links. Nodes have associated links between them. Nodes and links have name and type attributes. They may have a set of associated labels that store other descriptive information. Each label has an associated icon and text.

Entity-relationship models have been widely used in database design. The relational database schemas derived from these models are naturally in third normal form which is a desirable characteristic (Barker, 1989). Because of the explicit typing and the recognition of sub- and super-types, it is also straightforward to implement these models using object-oriented databases.

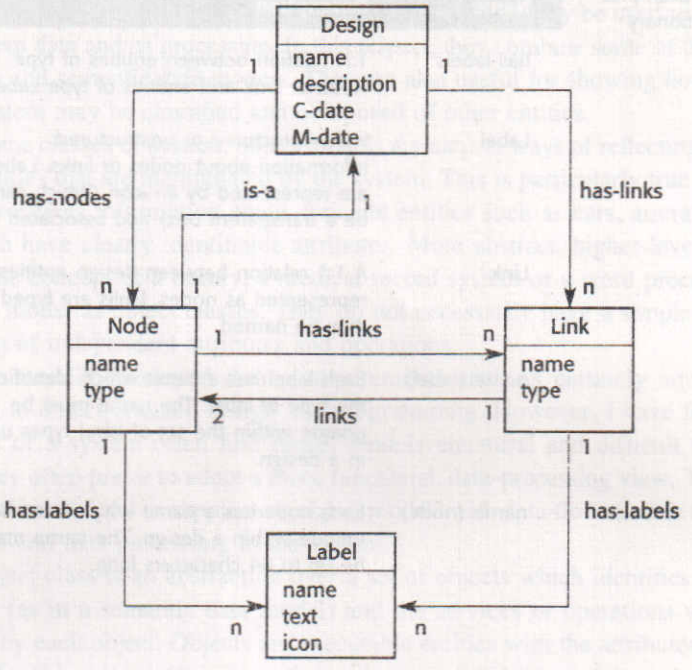


Figure 7.8 Semantic data model of a software design

Like all graphical models, ERA models lack detail and they should be supplemented with more detailed descriptions of the entities, relationships and attributes included in the model. These more detailed descriptions can be collected in a repository or data dictionary. Data dictionaries are generally useful when developing system models and may be used to manage all information from all types of system model.

A data dictionary is, simplistically, an alphabetic list of the names which are included in the different models of a system. As well as the name, the dictionary should include an associated description of the named entity and, if the name represents a composite object, there may be a description of the composition. Other information such as the date of creation, the creator, and the representation of the entity may also be included depending on the type of model which is being developed.

The advantages of using a data dictionary are:

1. It is a mechanism for name management. Many different people may have to invent names for entities and relationships when developing a large system model. These names should be used consistently and should not clash. The data dictionary software can check for name uniqueness and tell requirements analysts of name duplications.
2. It serves as a store of organisational information that can link analysis, design, implementation and evolution. As the system is developed, information is taken to inform the development. New information is added to it. All information about an entity is in one place.

Figure 7.9 Examples of data dictionary entries

Name	Description	Type	Date
has-labels	1:N relation between entities of type Node or Link and entities of type Label.	Relation	5.10.1998
Label	Holds structured or unstructured information about nodes or links. Labels are represented by an icon (which can be a transparent box) and associated text.	Entity	8.12.1998
Link	A 1:1 relation between design entities represented as nodes. Links are typed and may be named.	Relation	8.12.1998
name (label)	Each label has a name which identifies the type of label. The name must be unique within the set of label types used in a design.	Attribute	8.12.1998
name (node)	Each node has a name which must be unique within a design. The name may be up to 64 characters long.	Attribute	15.11.1998

All system names whether they be names of entities, types, relations, attributes or services should be entered in the dictionary. Support software should be available to create, maintain and interrogate the dictionary. This software might be integrated with other tools so that dictionary creation is partially automated. Most CASE tools which support system modelling also include support for data dictionaries.

An example of data dictionary entries is shown in Figure 7.9. I have used names taken from the semantic data model of a design shown in Figure 7.8. I have simplified the presentation of this example by leaving out some names and by shortening the associated information. A more complete data dictionary entry would include links to the representation of the information (e.g. a type declaration) and reverse links to where the name is used.

7.4 Object models

An object-oriented approach to the whole software development is now commonly used, particularly for interactive systems development. This means expressing the systems requirements using an object model, designing using objects and developing the system in an object-oriented programming language such as Java or C++.

Object models developed during requirements analysis may be used to represent both system data and its processing. In this respect, they combine some of the uses of data-flow and semantic data models. They are also useful for showing how entities in the system may be classified and composed of other entities.

For some classes of system, object models are natural ways of reflecting the real-world entities that are manipulated by the system. This is particularly true when the system processes information about concrete entities such as cars, aircraft, books, etc. which have clearly identifiable attributes. More abstract, higher-level entities, such as the concept of a library, a medical record system or a word processor, are harder to model as object classes. They do not necessarily have a simple interface consisting of independent attributes and operations.

Object models developed during requirements analysis certainly simplify the transition to object-oriented design and programming. However, I have found that end-users of a system often find object models unnatural and difficult to understand. They often prefer to adopt a more functional, data-processing view. Therefore, it is sometimes helpful to supplement object models with data-flow models that show the end-to-end data processing in the system.

An object class is an abstraction over a set of objects which identifies common attributes (as in a semantic data model) and the services or operations which are provided by each object. Objects are executable entities with the attributes and services of the object class. Objects are instantiations of the object class and many different objects may be created from a class. Generally, the models developed using analysis focus on object classes and their relationships.

Models of systems which are developed during requirements analysis should model real-world entities using object classes. They should not include details of the individual objects in the system. Various types of object models can be produced showing how object classes are related to each other, how objects are aggregated to form other objects, how objects interact with other objects and so on. All of these can add to our understanding of a system which is being specified.

The analysis process for identifying objects and object classes is recognised as one of the most difficult areas of object-oriented development. Object identification is basically the same for analysis and design. The methods of object identification covered in Chapter 12, which discusses object-oriented design, may be used. I concentrate here on some of the object models which might be generated during the analysis process.

Various methods of object-oriented analysis were proposed in the 1990s (Booch, 1994; Coad and Yourdon, 1990; Rumbaugh *et al.*, 1991). These methods have a great deal in common and three of the key developers (Booch, Rumbaugh and Jacobson) decided to integrate their approaches to produce a unified method (Rumbaugh *et al.*, 1999b). The Unified Modeling Language (UML) used in this unified method has become an effective standard for object modelling. UML includes notations for different types of system model. We have already seen use-case models and sequence diagrams in Chapter 5 and state machine models earlier in this chapter.

An object class in UML as illustrated in the examples in Figure 7.10 is represented as a vertically oriented rectangle with three sections:

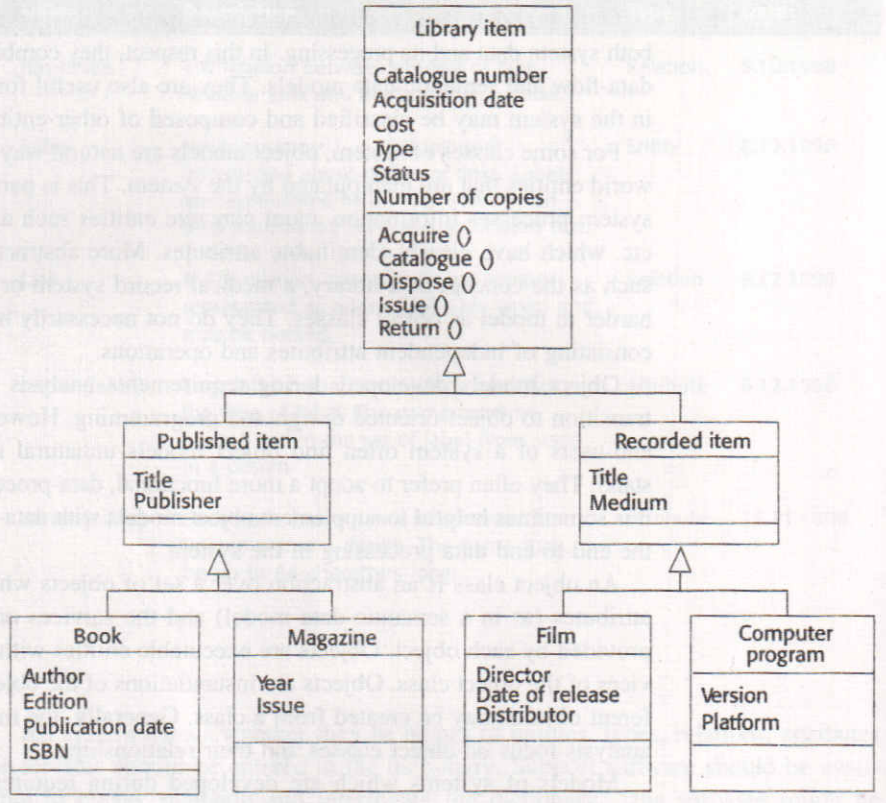


Figure 7.10 Part of a class hierarchy for a library system

1. The name of the object class is in the top section.
2. The class attributes are in the middle section.
3. The operations associated with the object class are in the lower section of the rectangle.

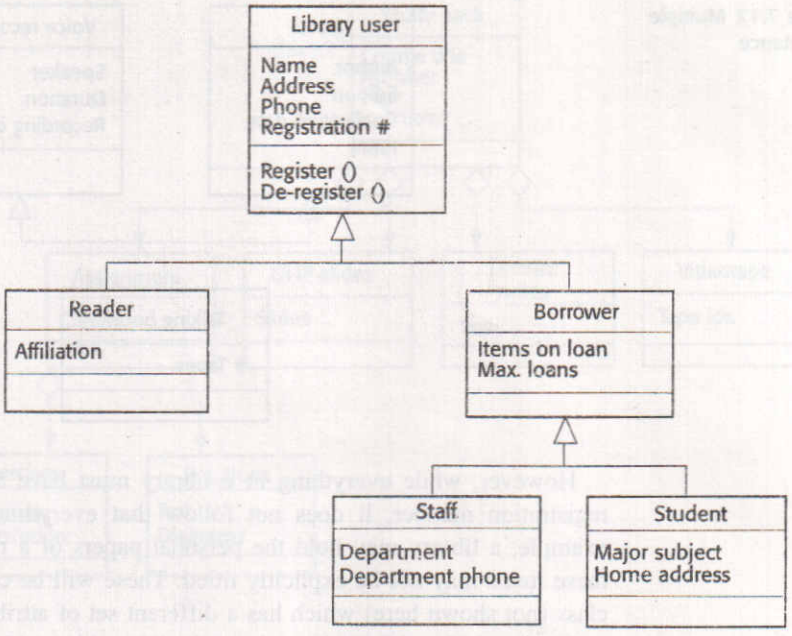
I don't have space to cover all of the UML so I focus here on object models for showing how objects can be classified and inherit attributes and operations from other objects, aggregation models which show how objects are composed and simple behavioural models which show object interactions.

7.4.1 Inheritance models

Object-oriented modelling involves identifying the classes of object which are important in the domain being studied. These are then organised into a taxonomy. A taxonomy is a classification scheme which shows how an object class is related to other classes through common attributes and services.

To display this taxonomy, the classes are organised into an inheritance hierarchy where the most general object classes are presented at the top of the hierarchy. More

Figure 7.11 User class hierarchy



specialised objects inherit their attributes and services. These specialised objects may have their own attributes and services.

Figure 7.10 illustrates part of a simplified class hierarchy for a library system model. This hierarchy gives information about the items held in the library. The library holds various types of item such as books, music, recordings of films, magazines, newspapers, etc. In Figure 7.10, the most general item is at the top of the tree and has a set of attributes and services which are common to all library items. These are inherited by the classes Published item and Recorded item which add their own attributes which are then inherited by lower-level items.

Figure 7.11 is an example of another inheritance hierarchy which might be part of a library model. In this case, the users of a library are shown. There are two classes of user: those who are allowed to borrow books and those who may only read books in the library without taking them away.

In the UML notation, inheritance is shown 'upwards' rather than 'downwards' as it is in some other object-oriented notations. That is, the arrowhead (shown as a triangle) points from the classes which inherit attributes and operations to the superclass. Rather than use the term inheritance, the UML refers to the generalisation relationship.

The design of class hierarchies is not easy. One advantage of developing such models is that the analyst needs to understand, in detail, the domain in which the system is to be installed. As an example of the subtlety of the problems which arise in practice, consider the library item hierarchy. It would seem that the attribute 'Title' could be held in the most general item, then inherited by lower-level items.

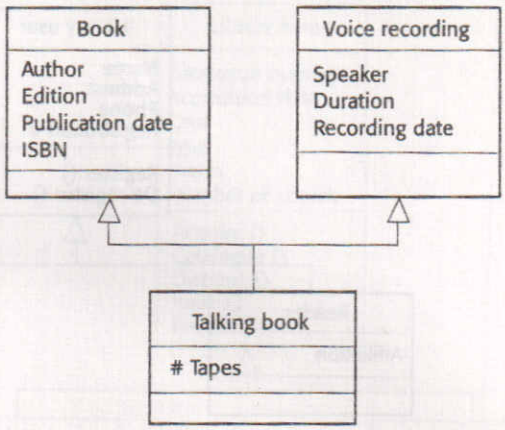


Figure 7.12 Multiple inheritance

However, while everything in a library must have some kind of identifier or registration number, it does not follow that everything must have a title. For example, a library may hold the personal papers of a retired politician. Many of these items may not be explicitly titled. These will be classified using some other class (not shown here) which has a different set of attributes.

Figures 7.10 and 7.11 show class inheritance hierarchies where every object class inherits its attributes and operations from a single parent class. Multiple inheritance models may also be constructed where a class has several parents. Its inherited attributes and services are a conjunction of those inherited from each superclass. Figure 7.12 shows an example of a multiple inheritance model which may also be part of the library model.

The main problem with multiple inheritance is designing an inheritance graph where objects do not inherit unnecessary attributes. Other problems include the difficulty of reorganising the inheritance graph when changes are required and resolving name clashes where two or more superclasses have attributes with the same name but different meanings. At the system modelling level, such clashes are relatively easy to resolve by manually altering the object model. They cause more problems in object-oriented programming.

7.4.2 Object aggregation

As well as acquiring attributes and services through an inheritance relationship with other objects, some objects are made up of other objects. That is, an object is an aggregate of a set of other objects. The classes representing these objects may be modelled using an object aggregation model as shown in Figure 7.13. In this example, I have modelled a library item which is a study pack for a university course. This study pack includes lecture notes, exercises, sample solutions, copies of transparencies used in lectures, and videotapes.

The UML notation for aggregation is to represent the composition by including a diamond shape on the source of the link. Therefore, Figure 7.13 can be read as

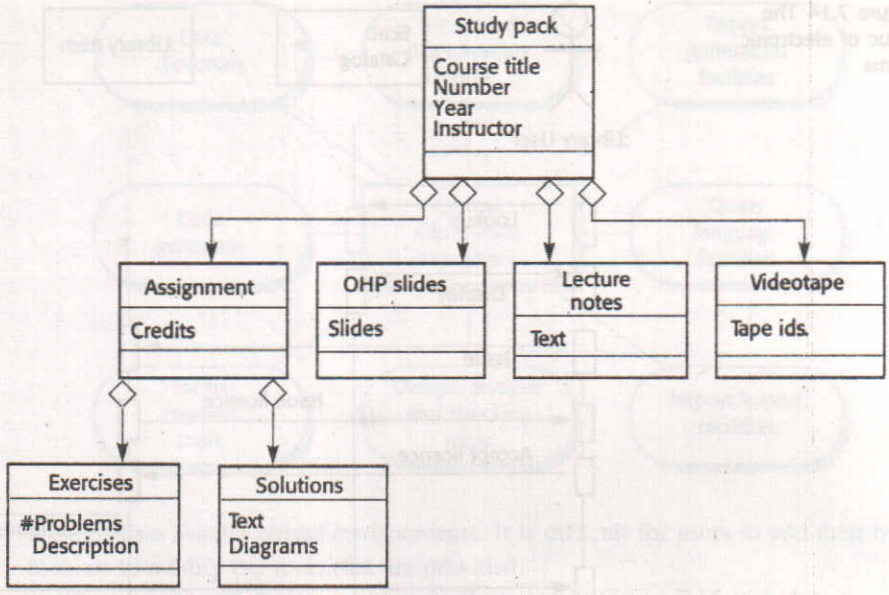


Figure 7.13 An aggregate object representing a course

'A study pack is composed of one of more assignments, OHP slide packages, lecture notes and videotapes'.

7.4.3 Object behaviour modelling

To model the behaviour of objects, we have to show how the operations provided by the objects are used. In the UML, behaviours are modelled using scenarios which are based on use-cases as discussed in Chapter 6. We have already seen an example of behaviour modelling in Chapter 6 where Figure 6.13 illustrates the management of a library catalogue. As well as sequence diagrams, the UML also includes collaboration diagrams which show the sequence of messages exchanged by objects. These are similar to sequence diagrams and I do not cover them here.

I illustrate how sequence diagrams may be used for behaviour modelling by describing a scenario where users withdraw items from the library in electronic form. For example, imagine a situation where the study packs shown in Figure 7.13 could be maintained electronically and downloaded to the student's computer.

Figure 7.14 shows a sequence diagram with objects along the top of the diagram. Operations are indicated by labelled arrows and the sequence of operations is from top to bottom. In this scenario, the library user accesses the catalogue to see if the item required is available electronically and, if so, requests the electronic issue of that item. For copyright reasons, this must be licensed so there is a transaction between the item and the user where the licence is agreed. The item to be issued is then sent to a network server object for compression before being sent to the library user.

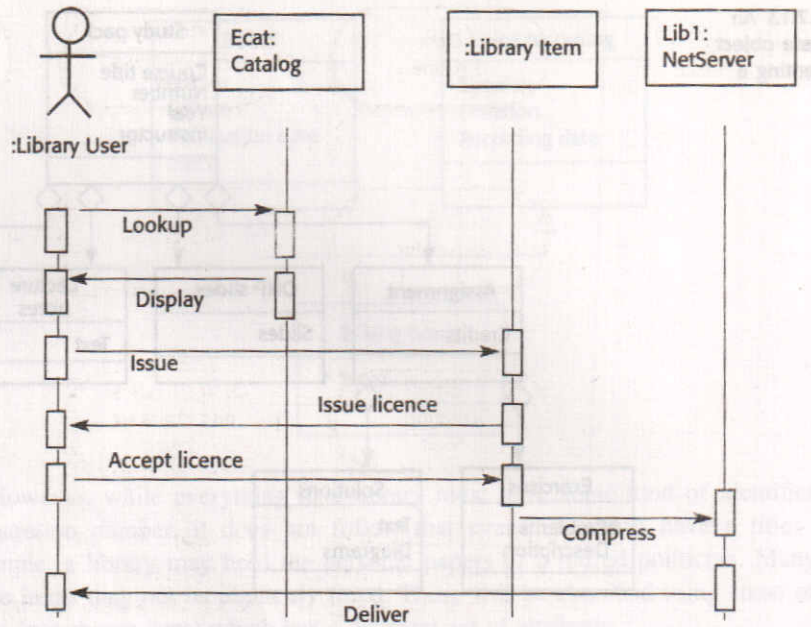


Figure 7.14 The issue of electronic items

7.5 CASE workbenches

A CASE workbench is a set of tools that supports a particular phase of the software process such as design, implementation or testing. The advantage of grouping CASE tools into a workbench is that tools can work together to provide more comprehensive support than is possible with a single tool. Common services can be implemented and called by all other tools. Workbench tools may be integrated either through shared files, a shared repository or shared data structures.

Analysis and design workbenches are designed to support system modelling during the analysis and design stages of the software process. These workbenches support the creation, editing and analysis of the graphical notations used in structured methods. Analysis and design workbenches may support a specific design or analysis method, such as object-oriented analysis and design. Alternatively, they may be more general diagram editing systems with knowledge of the diagram types that are used in several methods. Method-oriented workbenches incorporate method rules and guidelines. Some automatic checking of the diagrams is therefore possible.

Figure 7.15 shows the tools which may be included in an analysis and design workbench. These tools are normally integrated through a shared repository whose structure is proprietary to the workbench vendor. Analysis and design workbenches

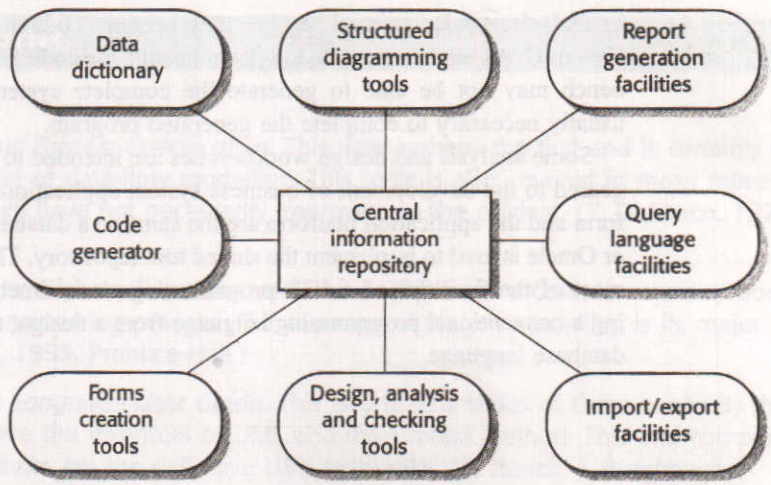


Figure 7.15
An analysis and design workbench

are therefore usually closed environments. It is difficult for users to add their own tools or to modify the tools that are provided.

The analysis and design workbench illustrated in Figure 7.15 includes:

1. *Diagram editors* to create data-flow diagrams, object hierarchies, entity-relationship diagrams, etc. These editors are not just drawing tools but are aware of the types of entities in the diagram. They capture information about these entities and save this information in the central repository.
2. *Design analysis and checking tools* which process the design and report on errors and anomalies. These may be integrated with the editing system so that user errors are trapped at an early stage in the process.
3. *Repository query languages* which allow the designer to find designs and associated design information in the repository.
4. *A data dictionary* which maintains information about the entities used in a system design.
5. *Report definition and generation tools* which take information from the central store and automatically generate system documentation.
6. *Forms definition tools* which allow screen and document formats to be specified.
7. *Import/export facilities* which allow the interchange of information from the central repository with other development tools.
8. *Code generators* which generate code or code skeletons automatically from the design captured in the central store.

In some cases, it is possible to generate a program or a program fragment from the information provided in the system model. The code generators which are included

in analysis and design workbenches may generate code in a language such as Java, C++ or C. As models exclude low-level details, the code generator in a design workbench may not be able to generate the complete system. Some hand-coding is usually necessary to complete the generated program.

Some analysis and design workbenches are intended to support methods that are geared to the development of business system applications. The development platform and the application platform are the same so a database system such as Sybase or Oracle is used to implement the shared tool repository. These workbenches include most of the facilities of a 4GL programming environment so, instead of generating a conventional programming language from a design, they may generate a 4GL database language.

KEY POINTS

- A model is an abstract view of a system which ignores some system details. Complementary system models can be developed which present different information about the system.
- Context models show how the system being modelled is positioned in an environment with other systems and processes. Architectural models, process models and data-flow models may be used as context models.
- Data-flow diagrams may be used to model the data processing carried out by a system. The system is modelled as a set of data transformations with functions acting on the data.
- State machine models are used to model a system's behaviour in response to internal or external events.
- Semantic data models describe the logical structure of the data which is imported to and exported by the system. These models show system entities, their attributes and the relationships in which they participate. They may be supplemented with data dictionaries that provide a more detailed description of the data.
- Object models describe the logical system entities and their classification and aggregation. They combine a data model with a processing model. Possible object models which may be developed include inheritance models, aggregation models and behavioural models.
- CASE workbenches support the development of system models by providing model editing, checking, reporting and documentation tools.

FURTHER READING

Structured Analysis and System Specification. This was perhaps the first and is certainly the best known description of data-flow modelling. This topic is also covered in many more modern books but they have not necessarily improved on the original. (T. DeMarco, 1978, Yourdon Press.)

Software Requirements: Objects, Functions and States. This book focuses on system modelling in the requirements engineering process. The coverage of system modelling is its major strength. (A. M. Davis, 1993, Prentice-Hall.)

The Unified Modeling Language User Guide. This is one of a series of three books by the same authors who were the inventors of UML and the Unified Method. This and companion texts by the same authors are the definitive UML textbooks. (G. Booch, J. Rumbaugh, I. Jacobson, 1998, Addison-Wesley.)

EXERCISES

7.1 Draw a context model for a patient information system in a hospital. You may make any reasonable assumptions about the other hospital systems which are available but your model must include a patient admissions system and an image storage system for X-rays.

7.2 Based on your experience with a bank ATM, draw a data-flow diagram modelling the data processing involved when a customer withdraws cash from the machine.

7.3 Model the data processing which might take place in an electronic mail system. You should model the mail-sending and mail-receiving processing separately.

7.4 Draw state machine models of the control software for:

- an automatic washing machine which has different programs for different types of clothes;
- the software for a compact disc player;
- a telephone answering machine which records incoming messages and displays the number of accepted messages on an LED display. The system should allow the telephone owner to dial in, type a sequence of numbers (identified as tones) and have the recorded messages replayed over the phone.

7.5 Extend the model of a software design shown in Figure 7.8 to include physical layout information. Nodes may be represented by composite symbols which are a combination of simpler symbols such as rectangle, line, etc. and are displayed at a particular coordinate. Links are made up of a number of line segments which may be solid or

dotted lines. Text is displayed in a specified font. The name associated with a node, link or label is positioned inside or near to the named item.

7.6 Model the object classes which might be used in an electronic mail system. If you have tried Exercise 7.3, describe the similarities and differences between the data processing model and the object model.

7.7 Using an entity-relation approach, describe a possible data model for the library cataloguing system described in this chapter. Assume that the items in the library are those modelled in Figure 7.10.

7.8 Develop an object model including a class hierarchy diagram and an aggregation diagram showing the principal components of a personal computer system and its system software.

7.9 Develop a sequence diagram showing the interactions involved when a student registers for a course in a university. Courses may have a limited number of places so the registration process must include checks that places are available. Assume that the student accesses an electronic course catalogue to find out about available courses.

7.10 Describe three modelling activities that may be supported by a CASE workbench for some analysis method. Suggest three activities that cannot readily be automated.