Chapter 3
System Modelling for Requirements Engineering

Art and science have their meeting point in method.

Edward Bulwer-Lytton, poet, 1803–1873 AD

3.1 Introduction

System modelling supports the analysis and design process by introducing a degree of formality into the way systems are defined. During system development it is often the case that pictures are used to help visualize some aspects of the development. Modelling provides a way of formalising these representations, through diagrams, by not only defining a standard syntax, but also providing a medium for understanding and communicating the ideas associated with system development.

The art of modelling is arguably the most creative aspect of the work of the systems engineer. There is no 'right' solution and models will evolve through various stages of system development. Models are most often represented visually and the information is therefore represented through connected diagrams. New methods such as object-orientation have advanced the concept of modelling, however most approaches are also based on the principles used and tested over time.

A good model is one which is easily communicated. They need to be used for communication within a development team, and also to an organisation as a whole including the stakeholders. The uses of a model can be diverse and cover a wide spectrum. It might be to model the activities of an entire organisation or to model a specific functional requirement of a system.

Modelling has the following benefits:

- Encourages the use of a *precisely defined vocabulary* consistent across the system.
- Allows system specification and design to be *visualized in diagrams*.
- Allows consideration of *multiple interacting aspects* and views of a system.
- Supports the *analysis of systems* through a defined discipline.
- Allows *validation* of some aspects of the system design through animation.

E. Hull et al., *Requirements Engineering*, DOI 10.1007/978-1-84996-405-0_3,

© Springer-Verlag London Limited 2011

- Allows *progressive refinement* towards detailed design, permitting *test case generation* and *code generation*.
- Encourages *communication between different organizations* by using common standard notations.

Much of the creativity and art of the systems engineer is expressed in the use of modelling techniques. This chapter considers a number of these representations and also some methods for Requirements Engineering that use them.

3.2 Representations for Requirements Engineering

3.2.1 Data Flow Diagrams

Data flow diagrams (DFDs) are the basis of most traditional modelling methods. They are the minimalist graphical representation of the system structure and interfaces and although initially produced for use in data representation and flow, the diagrams can in fact be used to show any type of flow, whether a computer-based system or not. The one output which DFDs do not show is that of control flow.

The elements in a data flow diagram consist of

- Data flows (labelled arrows)
- Data transformations (circles or “bubbles”)
- Data stores (horizontal parallel lines)
- External entities (rectangles)

The simple example in Fig. 3.1 shows the use of a data flow diagram in its traditional, information systems context.

Flows represent the information or material exchanged between two transformations. In real-world systems, this may be continuous, on demand, asynchronous etc. When using the notation, diagrams must be supported by textual descriptions of each process, data store and flow.

A *data dictionary* is used to define all the flows and data stores. Each leaf node bubble defines the basic functionality provided by the system components. These are described in terms of a *P-spec* or mini-spec. This is a textual description often written in a pseudo-code form.

The context diagram is the top-level diagram of a DFD and shows the external systems interacting with the proposed system, as in Fig. 3.2.

Bubbles can be decomposed another layer down. Each bubble is exploded into a diagram which itself may contain bubbles and data stores. This is represented in Fig. 3.3.

To illustrate the use of a DFD, consider an example of a context diagram for an Ambulance Command and Control system (Fig. 3.4). This is the starting point for a data-flow analysis of the system.

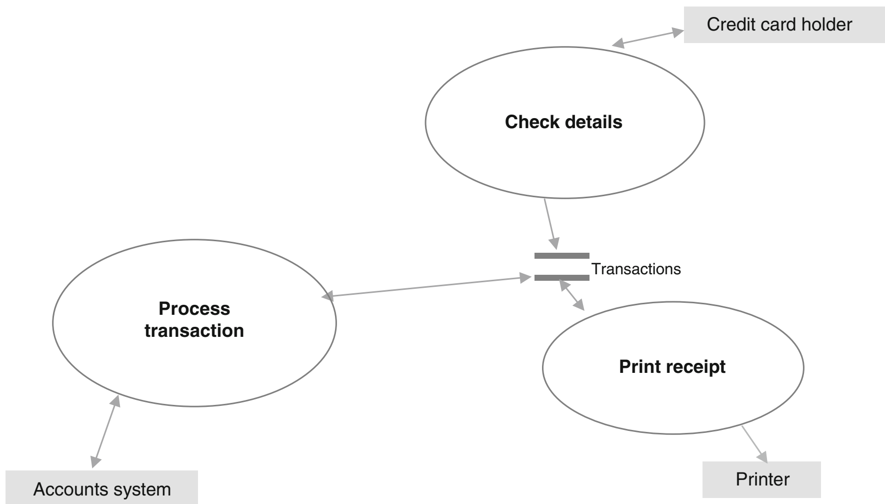

Fig. 3.1 Data flow diagram

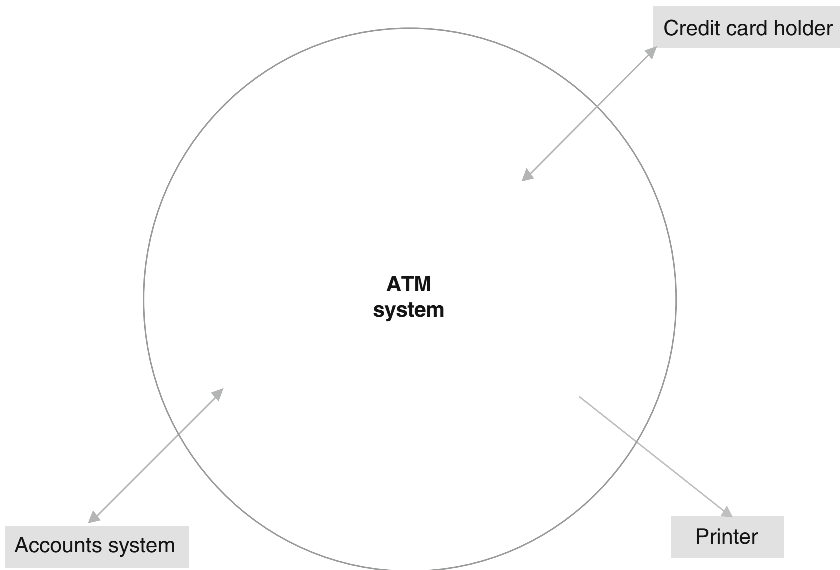

Fig. 3.2 Context diagram

The primary external entities are the *callers*, who make the emergency calls, and the *ambulances*, which will be controlled by the system. Note that *records* are an important output of the system (in fact a legal requirement) and a very important means of measuring "performance".

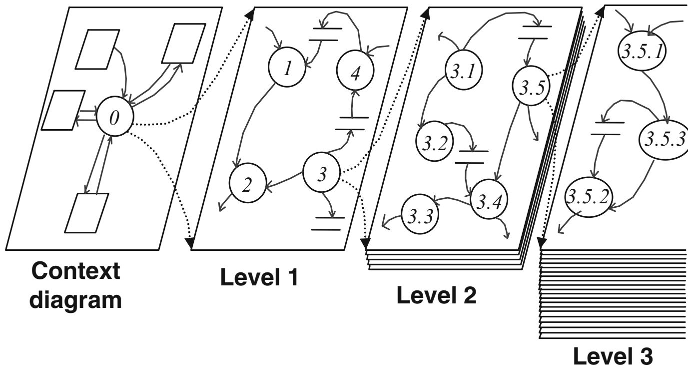

Fig. 3.3 Functional decomposition

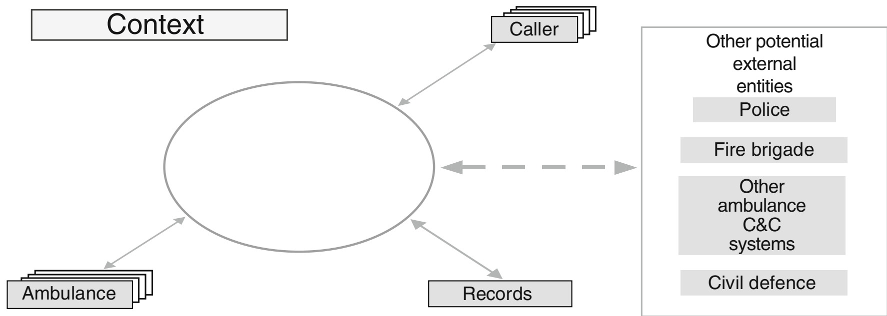

Fig. 3.4 Context diagram for Ambulance C&C System

Other potential external entities that would be required for a real system are shown in the diagram, but for simplicity we shall ignore them.

The next step is to identify the internal functionality of the system. Usually starting by drawing a function for each external entity as the minimal decomposition and then drawing the basic data that must flow between these top-level functions – see Fig. 3.5.

Following this, decomposition of the top-level functions takes place thus including more detail, as shown in Fig. 3.6.

The functional hierarchy in a set of data flow diagrams can be used as a framework for deriving and structuring system requirements. Figure 3.7 shows the functional structure for the Ambulance Command & Control example derived from Fig. 3.6.

Figure 3.7 also indicates some examples of requirements derived from this structure.

The hierarchical breakdown and interfaces give a good view of the component model, but they give a poor view of the "transactions" across the system i.e. from input to output (or to complete some system action) as can be seen in Fig. 3.8.

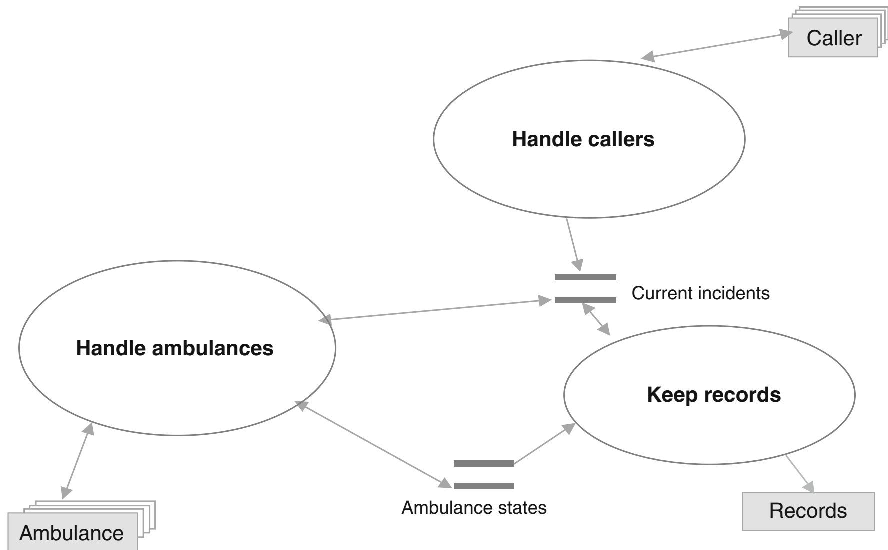

Fig. 3.5 Model for Ambulance C&C system

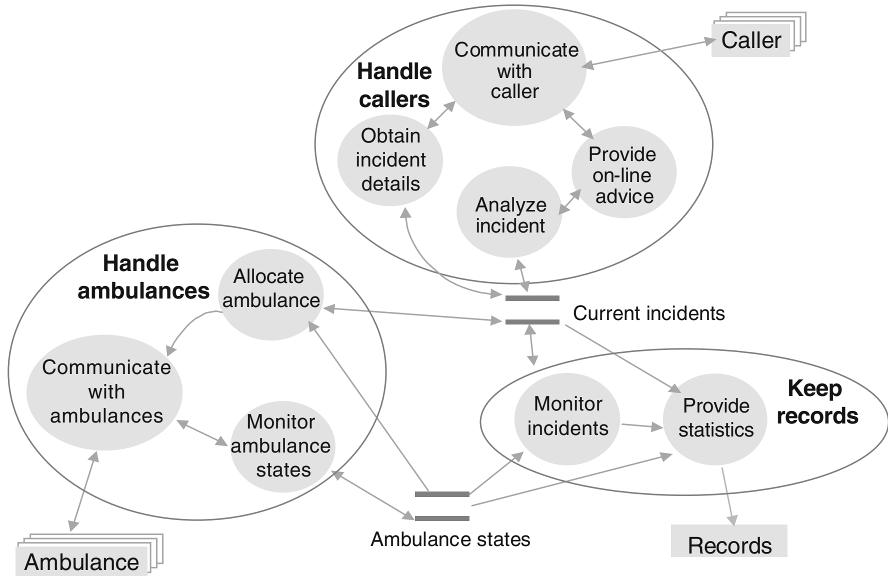

Fig. 3.6 Detailed model for Ambulance C&C system

It is therefore necessary to observe these transactions across the system in terms of the path(s) they follow, the time they take and the resources they absorb. Animating the stakeholder requirements and being able to see which functions are

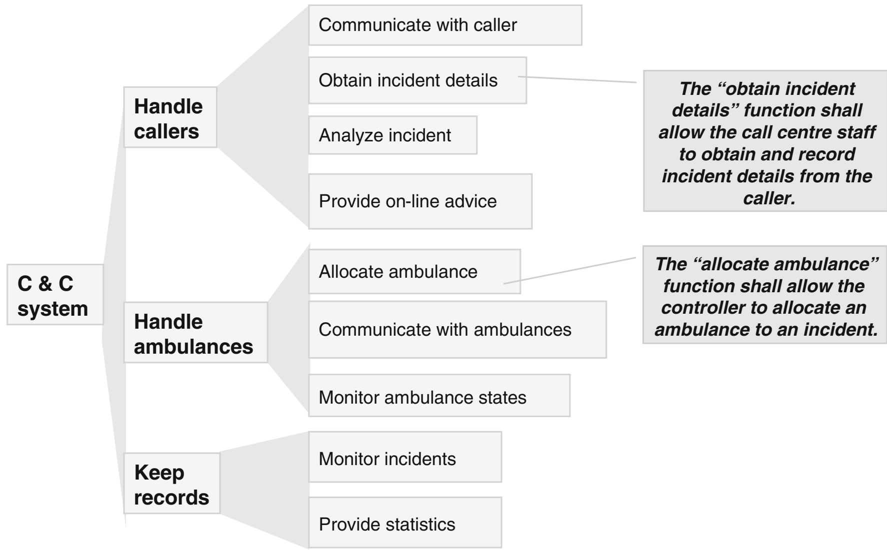

Fig. 3.7 Functional structure for Ambulance Command & Control system

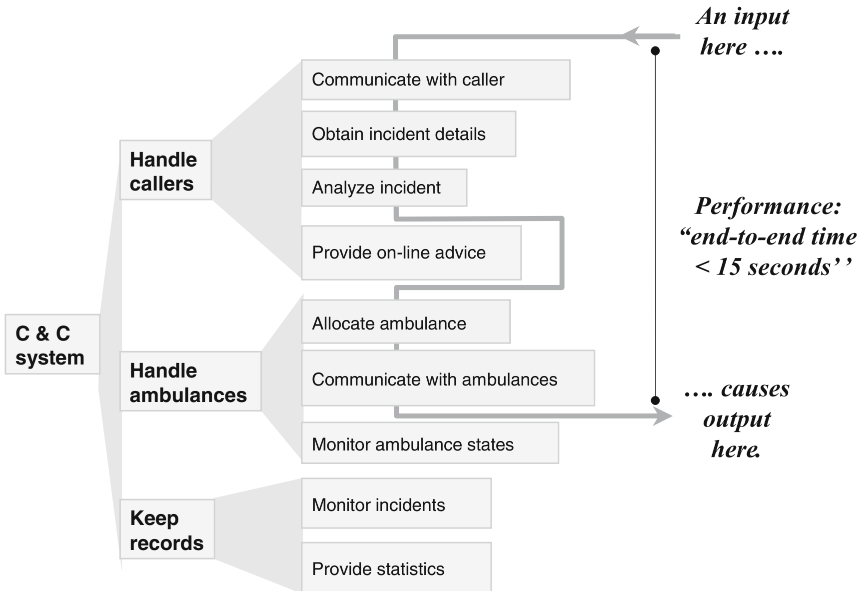

Fig. 3.8 System transactions

operating, will illustrate major transactions, but an alternative way of showing the system transactions is to mark them on to a data flow diagram as shown in Fig. 3.9, using the thick arrows.

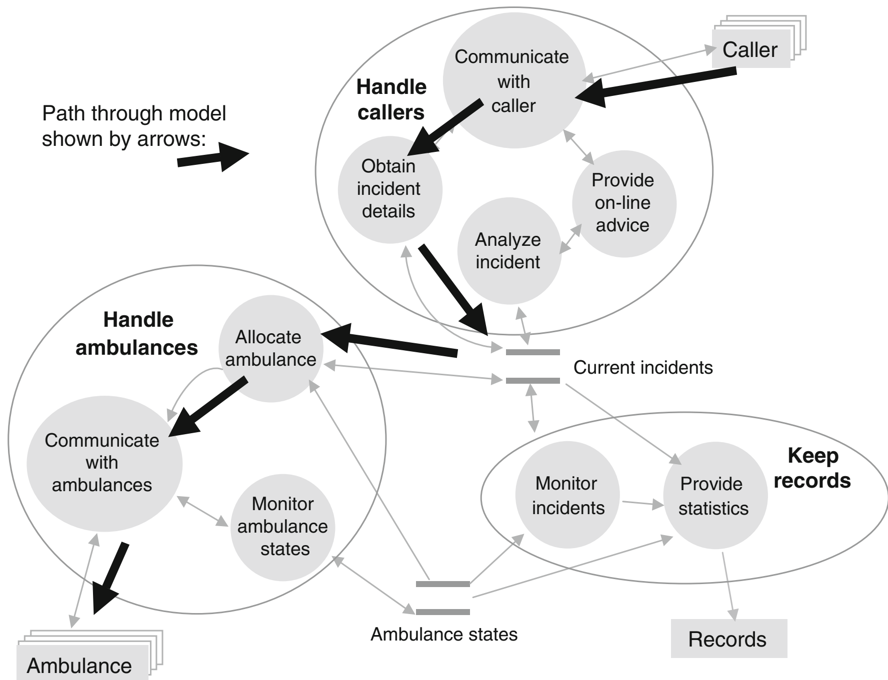

Fig. 3.9 System transactions for Ambulance Command & Control system

DFDs are good at presenting structures but they are not very precise. DFDs are less precise than text for developing a complete definition of a system – interface lines can mean anything, and single words can summarize anything. They cannot handle constraints properly.

A DFD clearly shows functions and interfaces. It can be used to identify end-to-end transactions, but does not directly show them. Ideally we would like to view the diagrams with an "expand in place" approach so that it is possible to view the context in which each level of decomposition is intended to work. Few CASE tools provide this level of facility.

Figure 3.6 actually breaks the conventions for drawing DFDs, because it shows a decomposition of the overall system into several processes and it also shows external agencies with which the system must interact. We advocate a pragmatic use of DFDs, rather than strict adherence to a conceptually pure ideal. To follow precisely the rules for drawing DFDs, the external agencies should appear only in the context diagram, and hence should not be visible at this level. However, the diagram would be far less meaningful if the external agencies were not shown and the flows to them left dangling (which is the defined convention for them).

In summary, DFDs:

- Show overall functional structure and flows.
- Identify functions, flows and data stores.

- Identify interfaces between functions.
- Provide a framework for deriving system requirements.
- Tools are available.
- Widely used in software development.
- Applicable to systems in general.

3.2.2 Entity-Relationship Diagrams

Modelling the retained information in a system, for example flight plans, system knowledge and data base records, is often important. Entity relationship diagrams (ERDs) provide a means of modelling the entities of interest and the relationships that exist between them. Chen (1976) initially developed ERDs. There is now a very wide set of alternative ERD notations.

An *entity* is an object that can be distinctly identified such as: customer, supplier, part, or product. A *property* (or *attribute*) is information that describes the entity. A *relationship* has cardinality, which expresses the nature of the association (one-to-one, one-to-many, many-to-many) between entities. A *subtype* is a subset of another entity, i.e. a type X is a sub-type of Y if every member of X belongs to Y.

ERDs define a partial model of the system by identifying the entities within the system and the relationships between them. It is a model that is independent of the processing which is required to generate or use the information. It is therefore an ideal tool to use for the abstract modelling work required within the system requirements phase. Consider the example Ambulance C&C system in Fig. 3.10.

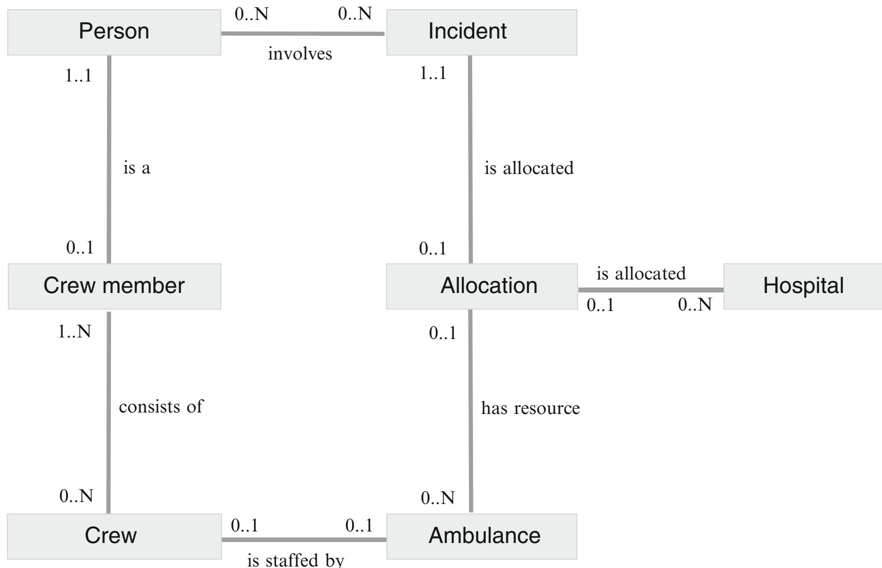

Fig. 3.10 ERD for Ambulance C&C system

3.2.3 Statecharts

Functionality and data flows are not enough for requirements definition. It is also necessary to be able to represent the behaviour of the system and in some circumstances consider the system as having a finite number of possible ‘states’, with external events acting as triggers that lead to transitions between the states.

To represent these aspects it is necessary to examine what states the system can be in and how it responds to events in these states. One of the most common ways of doing this is to use Harel’s Statecharts (Harel 1987).

Statecharts are concerned with providing a behavioural description of a system. They capture hierarchy within a single diagram form and also enable concurrency to be depicted and therefore they can be effective in practical situations where parallelism is prevalent. A labelled box with rounded corners denotes a state. Hierarchy is represented by encapsulation, and directed arcs, labelled with a description of the event, are used to denote a transition between states.

The descriptions of state, event and transition make statecharts suitable for modelling complete systems.

Figure 3.11 presents a statechart for an aircraft flight. The two top-level states are “Airborne” and “On Ground”, with defined transitions between them. Inside the “Airborne” state, there are three independent sets of states, while within the

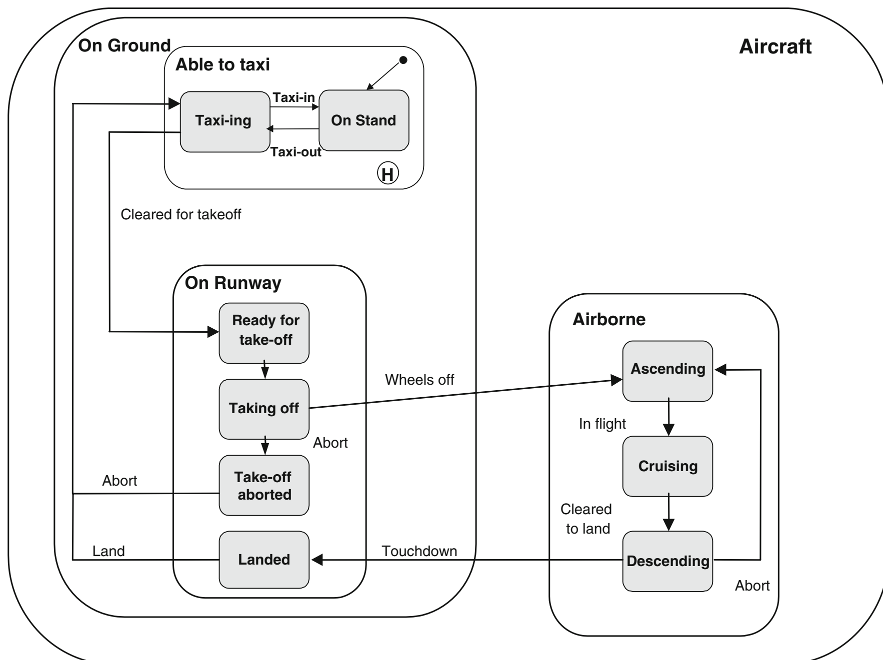

Fig. 3.11 Statechart for aircraft flight

"On Ground" state there are states for "Able to Taxi" and "On Runway". Inside the "On Ground" state, there are further states for 'taxiing' and 'on stand'.

The 'Airborne' state is entered when the aircraft wheels leave the ground and the 'On Ground' state is entered when the wheels touch down. Each of these states can now be further refined in a hierarchical way.

Statecharts introduce one further useful notion, that of history. When a state with the (H) annotation is re-entered, then the sub-state that was exited is also re-entered.

3.2.4 Object-Oriented Approaches

Object-orientation provides a rather different approach from that of the structured analysis approach. *Objects* describe stable (and hopefully) re-usable components.

Object-orientation tries to maximize this re-usability by asking the systems engineer to pick persistent objects i.e. those that can be used in system requirements and design.

So the goals of object-orientation are to:

- Encapsulate behaviour (states and events), information (data) and actions within the same *objects*.
- Try to define *persistent objects*, which can be used within both requirements and design phases.
- Add information by defining the *objects* in more detail.
- Create new objects by specialisation of existing *objects*, not creation of new objects.

Object-orientation focuses on the behaviour of objects, and their inter-relationships. A flat organization of objects is sometimes assumed, but this is not necessary, or even desirable. The analyst looks for entities that are long-lived, and models the behaviour of the system around them. This approach gives a coherent behavioural definition of the system. System elements should be re-usable because the elements (if not their behaviour) can be incrementally enhanced.

Some methodologists insist that design (and even implementation) is refinement of the analysis models. This can be a tall order for non-trivial systems. However, the progression from analysis, design to implementation is often far clearer in object-orientation than in other approaches. More analysis elements end up being represented in the implementation than is common in structured analysis and design. This is a tremendous aid to traceability and maintainability.

3.2.4.1 Class Diagrams

The class diagram is the basic diagramming notation from object-oriented analysis and design. Object-orientation arose out of computer-based simulation. The basic principle is that the contents of a software system should model the real world.

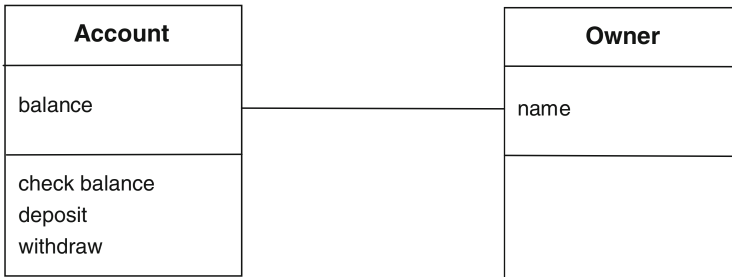

Fig. 3.12 Class diagram

The natural way to handle this is to have *objects* in the software that represent entities in the real world, both in terms of information and actions.

For example, in a banking system, instead of having an accounts file and separate accounts programs, there are *accounts objects* that have information such as *balance* and *overdraft limit* and relationships to other objects such as *account owner*. These objects have *operations* (also called *methods*) to handle the actions that are performed on accounts, like *check balance*, *deposit*, *withdraw*, etc.

The original reasoning behind this approach was that it made software development far more akin to modelling, and therefore more natural. As with many good ideas, practicalities intervene, and few object-oriented software systems can be seen as pure representations of the real world. Nevertheless, there is still considerable merit in the method.

A class (or object) diagram is shown in Fig. 3.12.

Class diagrams express information about classes of objects and their relationships. In many ways, they are similar to entity-relationship diagrams. Like them, they show how objects of a certain class relate to other objects of the same or different classes. The principal additional pieces of information are:

- Operations (methods)
- The concept of generalization
- Attributes within the objects

3.2.4.2 Use Cases

Use cases define the interaction that takes place between a user of a system (an actor) and the system itself. They are represented as process bubbles in a DFD type of context diagram. The use case diagram contains the actors and the use cases and shows the relationship between them. Each use case defines functional requirements for the system. Actors do not need to be human, even though they are represented as stick figures, but in fact represent *roles*. Each of the actors will have an association with at least one use case.

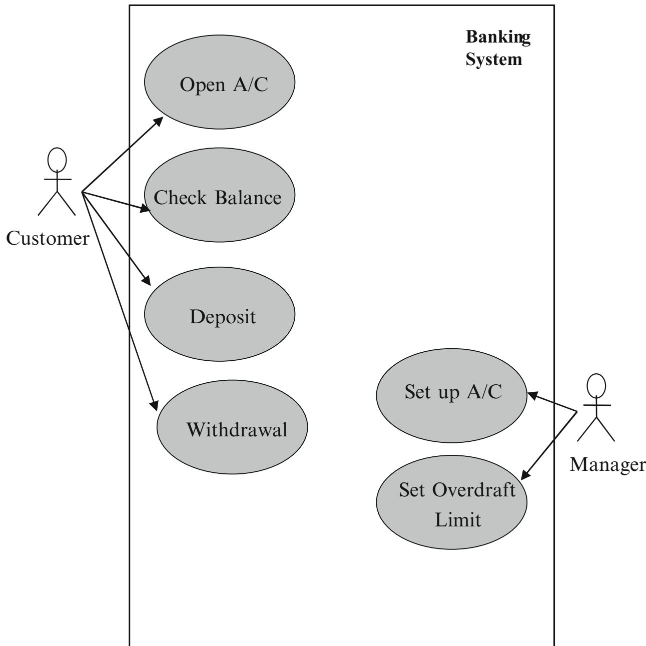

Fig. 3.13 Use case diagram for banking system

The system boundary is also defined on the use case diagram by a rectangle, with the name of the system being given within the box. Normally significant, and useful, textual information is associated with each use case diagram.

Figure 3.13 presents a use case diagram for a banking system.

3.3 Methods

A *method* is a degree more prescriptive than a modelling approach – it tells us what to do to and in what order to do it. Methods use various representations ranging from natural language, through diagrammatic forms to formal mathematics. Methods indicate when and where to use such representations. Those methods that use diagrammatic representations are usually referred to as 'structural methods'; those that use object-orientation are referred to as 'object-oriented methods' and those that use mathematics are referred to as 'formal methods'.

The purpose of the representations used in a method is to capture information. The information capture is aided by defining the set of concepts that a diagram represents, and the syntactic rules that govern the drawing of diagrams.

As we have seen in the earlier sections of this chapter, there are a variety of different representations used for system modelling. Most *methods* – Yourdon (1990), DeMarco (1978), Shlaer and Mellor (1998), Rumbaugh (1991), to name but a few,

are a reorganization of these concepts, varying the choice and the order in which they are done, often with minor enhancements. Interestingly, similarities between these methods are far more striking than their differences.

3.3.1 Viewpoint Methods

A viewpoint-based approach to Requirements Engineering recognises that requirements should not be considered from a single perspective. It is built on the premise that requirements should be collected and indeed organised from a number of different *viewpoints*. Basically two different kinds of viewpoint have been proposed:

- Viewpoints associated with stakeholders
- Viewpoints associated with organisational and domain knowledge

The role of the stakeholder is well understood in Requirements Engineering, however viewpoints associated with organisation and domain knowledge may be those associated with some aspect of security, marketing, database system, regulation, standard etc. Such viewpoints are not associated with a particular stakeholder, but will include information from a range of sources.

The following sections consider three different methods based on viewpoints.

3.3.1.1 Controlled Requirements Expression (CORE)

CORE was originally developed following work on requirements analysis carried out for the UK Ministry of Defence. A key finding of this work was that methods often started by defining the context of a solution to a problem, rather than attempting to define the problem itself, before beginning to assess possible solutions. CORE was specifically designed to address the latter approach. Figure 3.14 indicates the concepts and representations used in CORE.

The central concept of CORE is the viewpoint and the associated representation known as the viewpoint hierarchy. A viewpoint can be a person, role or organisation that has a view about an intended system. (This concept has been used as the basis of user viewpoint analysis by Darke and Shanks 1997). When used for system requirements the viewpoints can also represent the intended system, its subsystems and systems that exist within the environment of the system that may influence what the system must do. The viewpoints are organised in a hierarchy to provide a scope and also to guide the analysis process.

If we consider as an example, an aircraft brake and control system (ABCS), then Fig. 3.15 shows a possible list of initial viewpoints arrived at by means of brainstorming.

Having produced a list of potential viewpoints, they are organised into a hierarchy by grouping related candidates. Boundaries are drawn around related sets and this is repeated until all candidates have been enclosed and a hierarchy is produced.

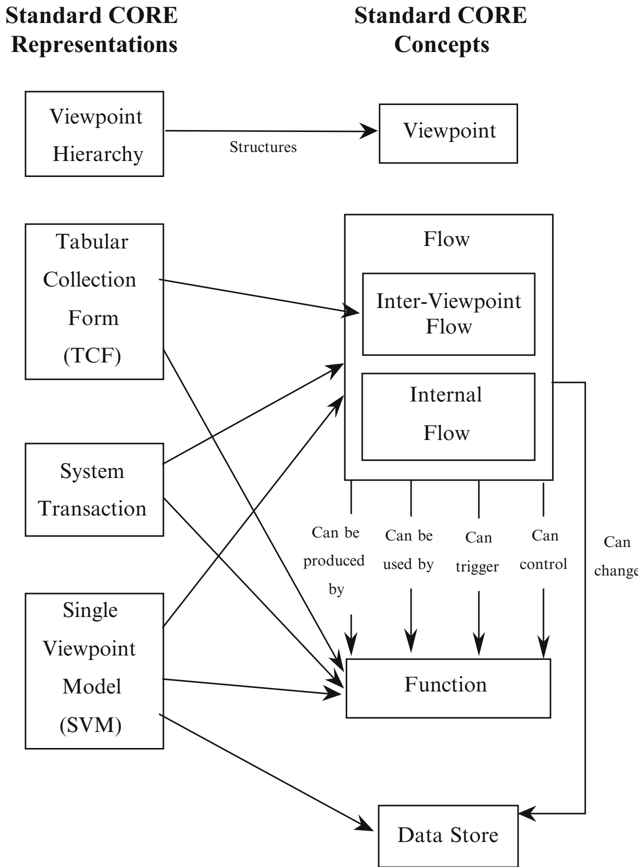

Fig. 3.14 Representations and concepts in CORE

Figure 3.16 shows a partial hierarchy for the aircraft braking control system.

In CORE the actions that each viewpoint must perform are determined. Each action may use or produce information or other items (e.g. commodities) relevant to the system in question. The information generated by the analysis is recorded in a Tabular Collection Form (TCF) as indicated in Table 3.1.

Lines are drawn between adjacent columns to indicate the flows that take place.

Once each viewpoint has been analysed in this way, the TCFs at each level in the viewpoint hierarchy are checked as a group to ensure that the inputs which each viewpoint expects are generated by the source viewpoint and that the outputs which each action generates are expected by the viewpoint(s) indicated as the destination(s) for them.

Returning to the example aircraft braking control system, part of the TCF for the system is shown in Table 3.2.

Further analysis consists of developing a more detailed data flow model for each viewpoint in turn. The starting point for these Single Viewpoint Models (SVMs) is

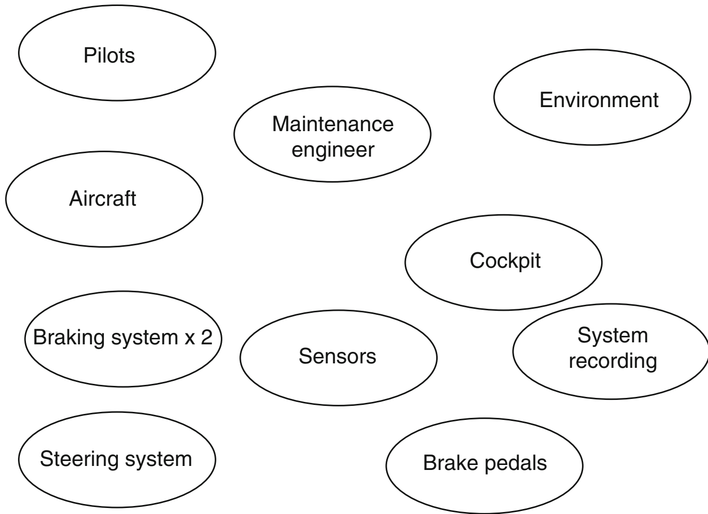

Fig. 3.15 Initial viewpoints for ABCS

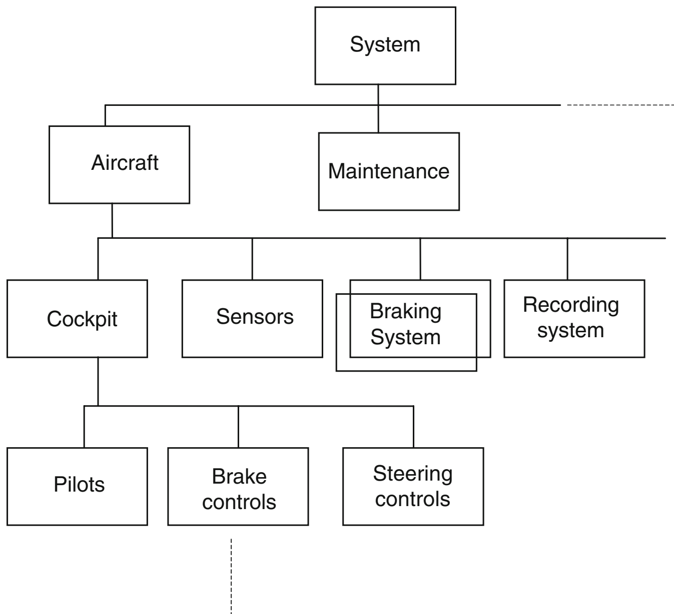

Fig. 3.16 Hierarchy example

the information recorded in the TCFs. SVMs add flows that are entirely within a viewpoint and data stores. The SVMs also define how actions are controlled and triggered by flows from other actions.

Thus the analysis is driven top-down by analysing each stratum in the viewpoint hierarchy. With top-down analysis, it can be difficult to know when to stop and to

Table 3.1 Tabular collection form

Source	Input	Action	Output	Destination
The viewpoint from which the input comes	*The name of the input item*	*The action performed on one or more inputs to generate required outputs*	*The name(s) of any outputs generated by the action*	*The viewpoint to which the output is sent*

Table 3.2 TCF example

Source	Input	Action	Output	Destination
Channel 1,2	Power on of channel 1,2		Self test ok	Channel 1,2
			Self test fail	
Cockpit	Power up	Power up self test	Channel fault	
			NWS isolator valve fault	
			Autobrake fault	System recording
			Shutoff valve fault	
			Towing state	
Other sensors/ actuators	Towing controlled	Monitor towing	Towing control on	Aircraft
			Towing control off	
Channel 1,2	Operational of channel 1,2			
Wheel speed	Wheel speeds	Monitor wheel speeds	Speed > 70 knots	Cockpit

predict where the analysis will lead. The approach of first identifying the viewpoints and then using them to control the subsequent analysis provides a controlled way of doing analysis in a top-down manner. This overcomes a major problem associated with data flow based analysis. This element of control is alluded to in Controlled Requirements Expression, the full name of CORE.

The other main concept of CORE is the *system transaction*. This is a path through the system from one or more inputs, data flows or events to one or more specific output flows or events. The system transactions address how a system is intended to operate. They provide a view orthogonal to the top-down analysis. System transactions provide a sound basis for discussing the non-functional requirements.

3.3.1.2 Structured Analysis and Design Technique (SADT)

SADT is a method of structured analysis, based on the work undertaken by Ross on Structured Analysis (SA) in the 1970s (Ross 1977). It is graphically oriented and adopts a purely hierarchical approach to the problem with a succession of blueprints both modularising and refining it until a solution is achieved. The basic element of SADT is the box, which represents an activity (in activity diagrams) or data (in data diagrams). The boxes are joined by arrows representing either the data needed or provided by the activity represented by the box (in activity diagrams), or the process providing or using the data (in data diagrams).

There are four basic *arrows* associated with a *box*, as shown in Fig. 3.17. The type of arrow is implied by its point of connection to the box:

- *Input* arrows enter the box from the left side, and represent data that is available to the activity represented by the box.
- *Output* arrows exit the box from the right side, and represent data that is produced by the activity represented by the box i.e. the input data has been transformed by the activity represented by the box to produce this output.
- *Control* arrows enter the box from the top, and govern the way in which the transformation takes place.
- *Mechanism* arrows enter the box from below and control the way the activity may use outside mechanisms e.g. a specific algorithm or resources.

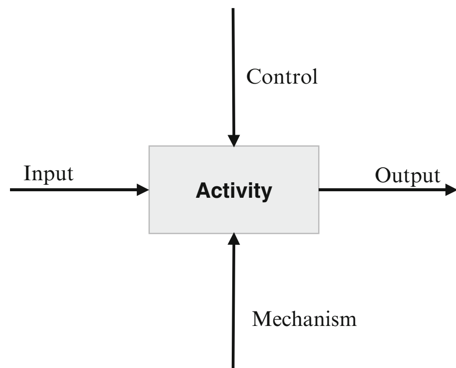

Fig. 3.17 SADT *box* and *arrows*

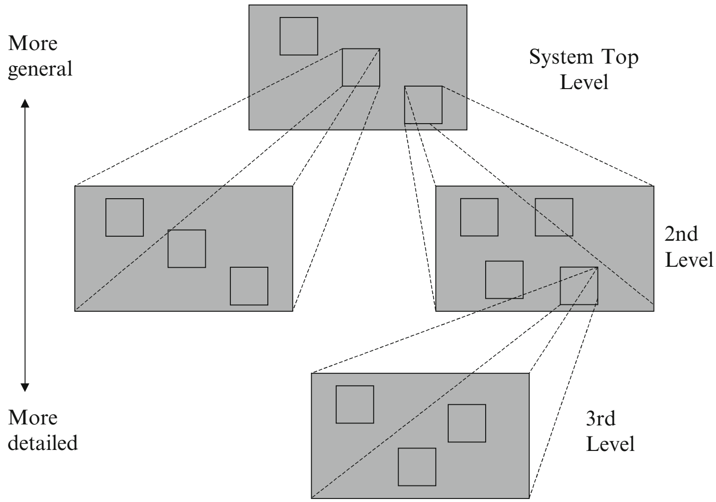

Fig. 3.18 Decomposition using SADT diagrams

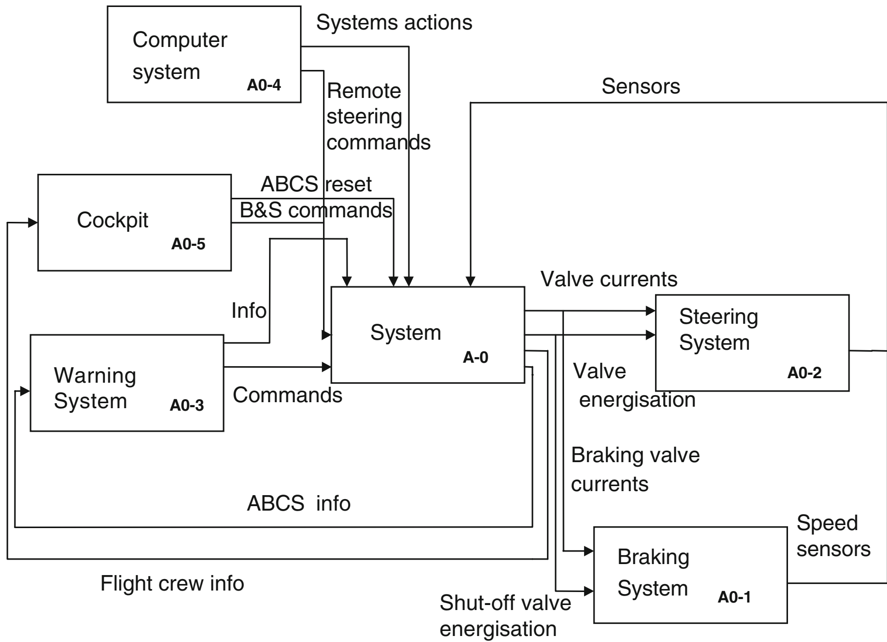

Fig. 3.19 SADT example

An SADT diagram is made up of a number of boxes with the associated set of arrows. A problem is refined by decomposing each box and generating a hierarchical diagram, as shown in Fig. 3.18.

Figure 3.19 shows an example activity diagram for an ABCS. This decomposition proceeds until there is sufficient detail for the design to proceed.

3.3.1.3 Viewpoint-Oriented Requirements Definition (VORD)

VORD (Kotonya 1996) is a method based on viewpoints. The model used is a service-oriented one, where the viewpoints are considered to be clients, if one was to think of it as a client-server system.

A viewpoint in VORD receives services from the system and in turn passes control information to the system. The service-oriented approach makes VORD suited for specifying interactive systems.

There are two types of viewpoint in VORD – **direct** and **indirect**:

- **Direct viewpoints** receive services from the system and send control information and data to the system.
- **Indirect viewpoints** do not interact directly with the system but rather have an 'interest' in some or all of the services delivered by the system.

There can be a large variation of indirect viewpoints. Examples include engineering viewpoints concerned with aspects to be undertaken by the systems engineer; external viewpoints which may be concerned with aspects of the system's environment; organisation viewpoints which may be concerned with aspects of safety etc.

There are three main iterative steps in VORD:

1. Viewpoint identification and structuring
2. Viewpoint documentation
3. Viewpoint requirements analysis and specification

The graphical notation for a viewpoint is shown in Fig. 3.20. A viewpoint is represented by a rectangle, which contains an identifier, label and type. Viewpoint attributes are represented by labels attached to a vertical line dropping down from the left-hand side of the rectangle.

The VORD method guides the systems engineer in identifying viewpoints. It provides a number of abstract viewpoints which act as a starting point for identification. See Fig. 3.21. (Following the convention for VORD diagrams, direct viewpoints are unfilled rectangles and indirect viewpoints are in greyscale). This class hierarchy is then pruned to eliminate viewpoint classes which are not relevant to a particular problem. The system stakeholders, the viewpoints representing other systems and the system operators are then identified. Finally, for each indirect viewpoint that has been identified consideration is given to who might be associated with it.

Based on this approach, Fig. 3.22 gives the viewpoints for a Pay & Display Car Park System.

'Cash User' and 'Credit Card User' viewpoints are specialisations of the 'Car Park Customer' viewpoint. 'Cash Collector' and 'Car Park Manager' are specialisations of

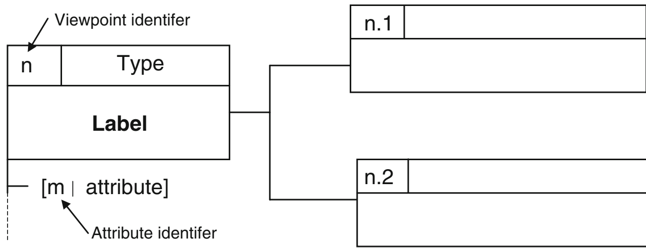

Fig. 3.20 Viewpoint notation

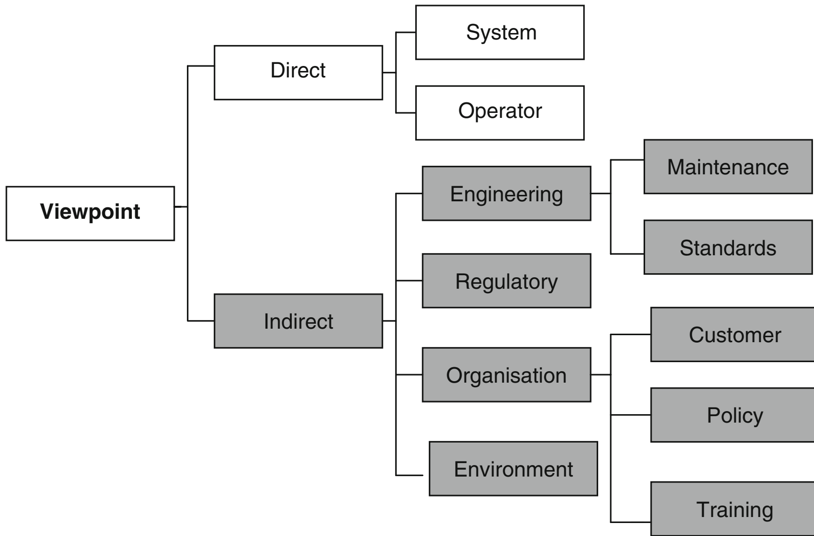

Fig. 3.21 Viewpoint classes

'Car Park Staff'. The 'Ticket Issuing' viewpoint represents the database of the organisation responsible for issuing the pay & display tickets. The 'Credit Card Database' is external and holds details of the customer's credit card details.

The next step in VORD is to document each of the viewpoint requirements. An example of how this is achieved is given in Table 3.3 which shows the initial viewpoint requirements for the 'Car Park Customer' viewpoint. The requirement type refers to a service (sv) or to a non-functional requirement (nf).

VORD also allows for attributes of viewpoints to be provided which characterise the viewpoint in the problem domain. These are important as they provide the data on which the system operates. As stated previously, these are represented on the viewpoint diagram by labels attached to a vertical line dropping down from the left-hand side of the rectangle as shown in Fig. 3.23.

System behaviour is modelled using event scenarios. These describe how the system interacts with the environment and provide a way of describing the complex interactions between the various viewpoints and the system.

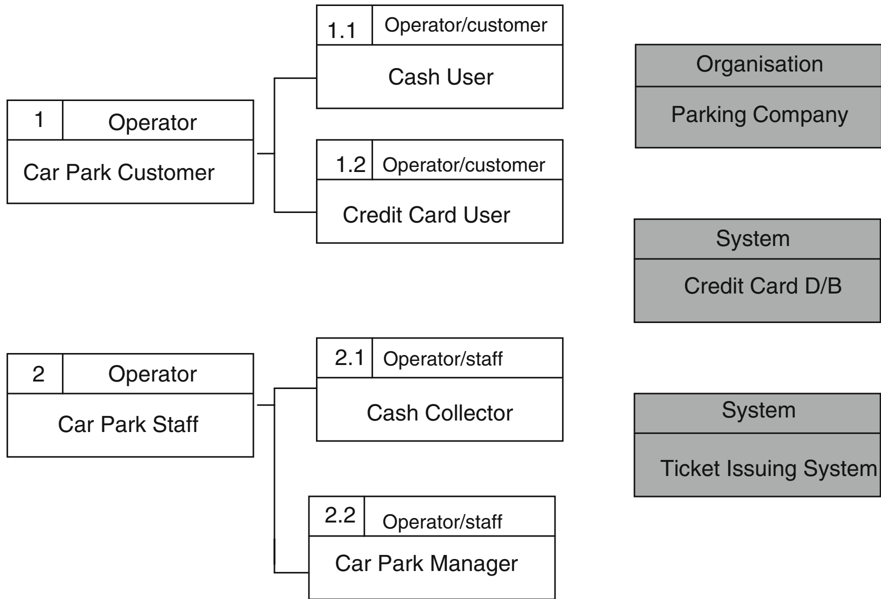

Fig. 3.22 Pay-and-display machine viewpoints

Table 3.3 Requirements from the car park customer viewpoint

Viewpoint			Requirement	
Identifier	Label		Description	Type
1	Customer	**1**.1	Provide facility for ticket based on suitable payment and length of stay	sv
1.1	Credit card user	**1.1**.1	Provide facility based on valid credit card	sv
		1.1.2	Provide ticket issuing service for customer	sv
		1.1.3	Ticket issuing service should be available 99/100 requests	nf
		1.1.4	Ticket issuing service should have a response time of no more than 30 s	nf
1.2	Cash user			

The final stage of VORD is to translate the results of the requirements analysis process into a requirements document, based on an industry standard.

3.3.2 *Object-Oriented Methods*

During the late 1980s and early 1990s numerous object-oriented methods emerged proposing different approaches to object-oriented (O-O) analysis and design. The earliest uses of O-O methods were those companies where time to market and

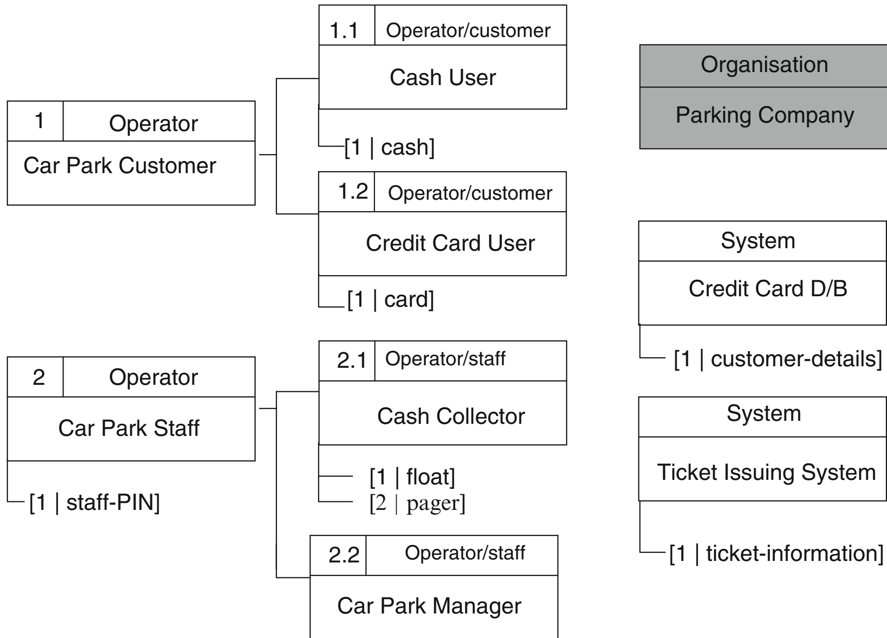

Fig. 3.23 Representation of viewpoint attributes

resistance to change were paramount. They included telecommunications, financial organisations and later aerospace, health care, banking, insurance, transportation etc.

The main players were Object-Oriented Analysis (OOA), Object Modelling Technique (OMT), Booch, and Objectory. Shlaer-Mellor was also there, but would not have been regarded as a truly O-O method. However it did play an important role in assisting in the identification of objects.

3.3.2.1 OOA

Object-oriented analysis (OOA) was developed by Coad and Yourdon (1991a). OOA is spread across three layers, as they are called. The first layer is the subject layer, which is concerned with object identification. Here the users are able to simply represent their understanding of the problem domain by identifying relevant problem domain objects. The second layer, called the attributes layer, is concerned with identifying attributes (data elements) associated with problem domain objects. The third and final layer is the services layer. This specifies the services (or operations) performed by each object.

In effect, OOA helps the systems engineer in identifying the requirements of a system, rather than how the software should be structured or implemented. It therefore describes the existing system, its operation and how the software system should interact with it.

3.3.2.2 OMT

The OMT method was developed by Rumbaugh. It aims to construct a series of object models that refine the system design until the final model is suitable for implementation. The approach is achieved in three phases. The analysis phase produces models of the problem domain. Three types of model are produced – the object model, the dynamic model and the functional model. The object model is the first one to be built. It uses notation similar to that used in OOA, which is based on the concept of ER modelling which describes the objects, their classes and the relationships between the objects. The dynamic model represents the behaviour of the system and uses an extension of Harel's statecharts. The functional model describes how the system functions are performed through the use of DFDs.

These models are arrived at by using an iterative approach. The design phase then structures the model and the implementation phase takes into account the appropriate target language constructs. In this way OMT covers not only the requirements capturing phase but also helps to inform the architectural design process.

3.3.2.3 Booch

The Booch method is one of the earliest O-O methods proposed. Although the method does consider analysis, its strength lies in the contribution it makes to the design of an object-oriented system. The approach is both incremental and iterative and the designer is encouraged to develop the system by looking at both logical and physical views of the system.

The method involves analysing the problem domain to identify the set of classes and objects and their relationships in the system. These are represented using a diagrammatical notation. The notation is extended further when considering the implementation of classes and objects and the services they provide. The use of state transition diagrams and timing diagrams are also an important part of this method.

3.3.2.4 Objectory

Jacobson proposed the Objectory method. Many of its ideas are similar to other O-O methods, but the fundamental aspect of this method is the scenario or *use case*, as described earlier in this chapter. The system's functionality should therefore be able to be described based on the set of use cases for a system – the use case model.

This model is then used to generate a domain object model, which can become an analysis model by classifying the domain objects into three types: interface objects, entity objects and control objects. This analysis model is then converted to a design model, which is expressed in terms of blocks, from which the system is implemented.

3.3.2.5 The UML

The Unified Modelling Language (UML) (OMG 2003) was an attempt to bring together three of the O-O approaches which had gained greatest acceptance – Booch, OMT and Objectory. In the mid-1990s Booch, Rumbaugh and Jacobson joined Rational to produce a single, common and widely usable modelling language. The emphasis was very much on the production of a notation rather than a method or process.

Since its inception the UML has undergone extensive development and change with various versions being launched. UML 1.0 became a standard in 1997 following acceptance by the Object Management Group (OMG). Version 1.3 was released in 1999 and in 2003 the UML 2.0 was released, which is the version used in this book. A discussion of the UML is provided in the following section.

3.3.3 The UML Notation

The UML is made up of a number of models, which together describe the system under development. Each model represents distinct phases of development and each will have a separate purpose. Each model is comprised of one or more of the following diagrams, which are classified as follows:

- Structure diagrams
- Behaviour diagrams
- Interaction diagrams

The 13 diagrams of UML2 are shown in Fig. 3.24 and represent all the diagrams which are available to the systems engineer. In reality many will not be used and often only a small subset of the diagrams will be necessary to model a system. Class diagrams, use case diagrams and sequence diagrams are probably the most frequently used. If dynamic modelling is required then activity diagrams and state machine diagrams should be used.

It is how the UML diagrams contribute to modelling which is of interest to us. The purpose of this section is not so much to provide an overview of UML2, but rather to show how models can be used in various aspects of Requirements Engineering.

Consider the banking example used earlier in this chapter. The class is the basic modelling diagram of the UML. Figure 3.25 presents a UML class diagram extending the set of classes to include ‘Account’, ‘Owner’, ‘Current Account’ and ‘Issued Cheque’ – used to model the system. As shown, each class has an associated set of attributes and operations, i.e. the relationships (in this case, generalisation and association) which exist between one or more classes.

Figure 3.26 gives a different example, that of a Baggage Handling System. This considers the stakeholder requirements which are firmly within the problem domain. When modelling, it is often the case that there are external systems, or

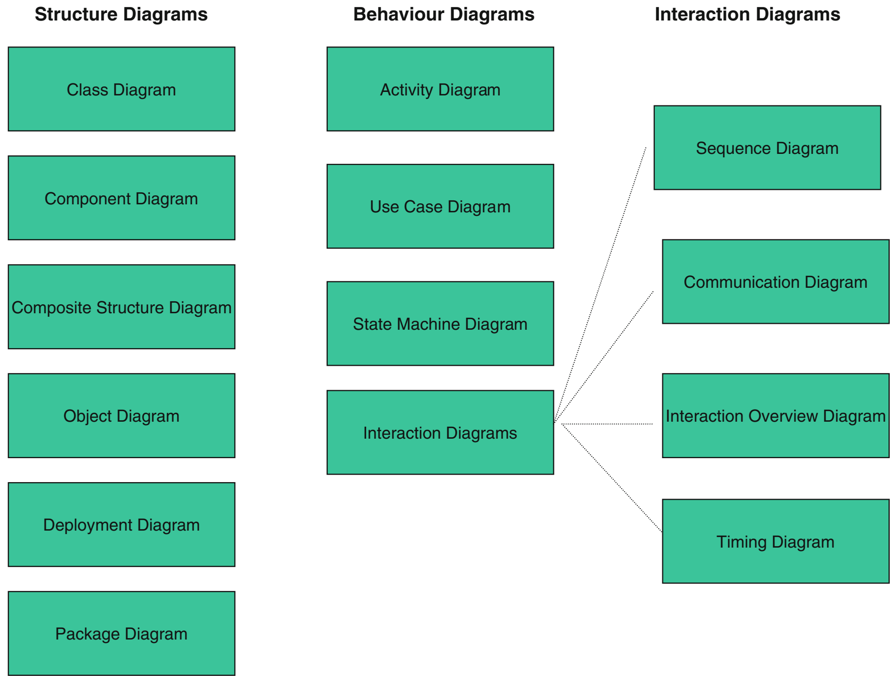

Fig. 3.24 UML diagrams

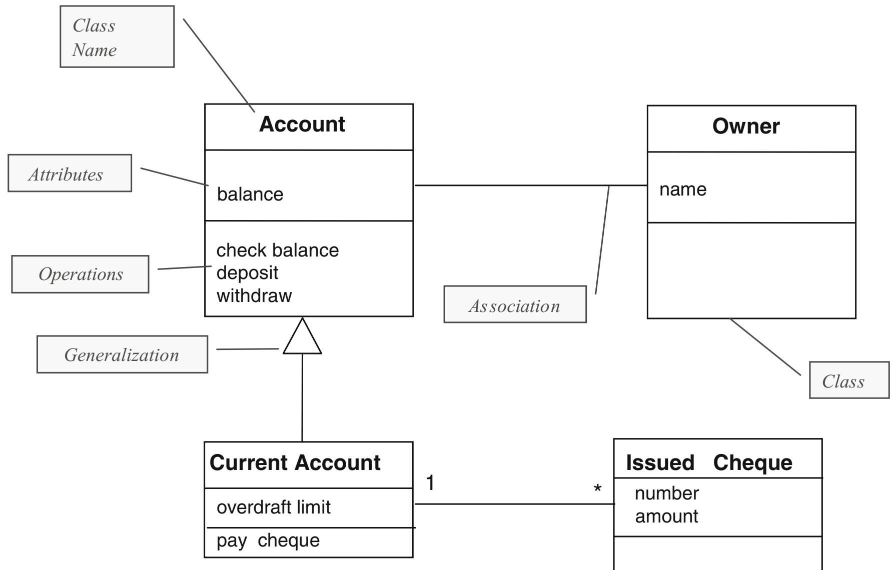

Fig. 3.25 Extended UML class diagram

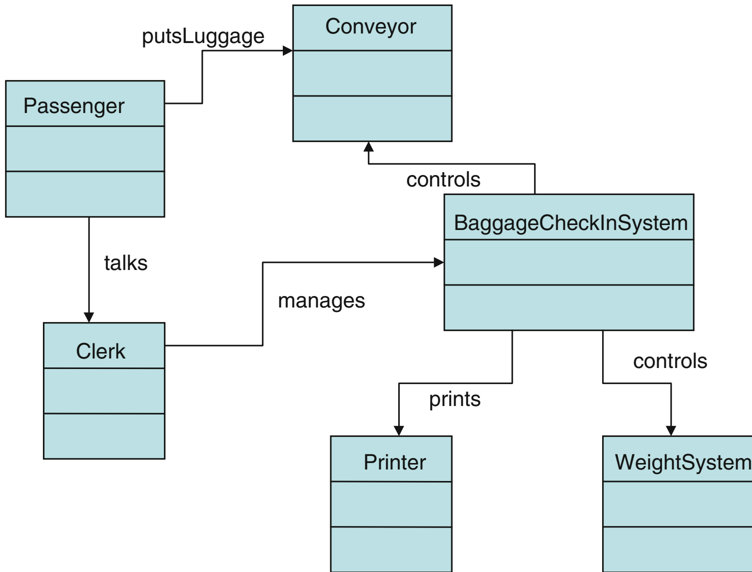

Fig. 3.26 Class diagram for Baggage Handling System

perhaps, devices which the system will use. These can be represented by classes. For the Baggage Handling System, classes are identified such as 'Passenger, 'Clerk', 'Conveyor' etc and also two embedded systems – **BaggageCheckInSystem** and **WeightSystem**. The associations between the systems and other classes serve to define aspects of the system context.

If we turn to the solution domain, then it becomes necessary to reason about function and behaviour. The class diagram therefore needs to be elaborated in order to show these attributes which will be necessary for modelling the system requirements. This is shown in Fig. 3.27.

Use case modelling is used to describe the functional requirements of systems. For our example we will consider two use case diagrams – one for the Baggage Handler System and one for the Baggage Check-in System. Figure 3.28 shows the first of these portrayed as the top-level system. Figure 3.29 is the use case diagram for the Baggage Check-in System. Both diagrams identify their respective system boundaries (marked by a rectangle) and identify the various stakeholders or actors which lie outside the system boundary. It should be noted that the highest level goals of the stakeholders are represented by the use cases. The «include» relationship shows that a use case is included in another use case, indicating the start of hierarchical decomposition.

The UML also provides diagrams to allow the systems engineer to model functionality and behaviour. A sequence diagram shows the interaction and collaboration which exists between objects and thus can model complex behaviour. It is depicted by messages which flow between objects over time. Figure 3.30 shows a sample sequence diagram. The objects are represented by rectangles at the top of

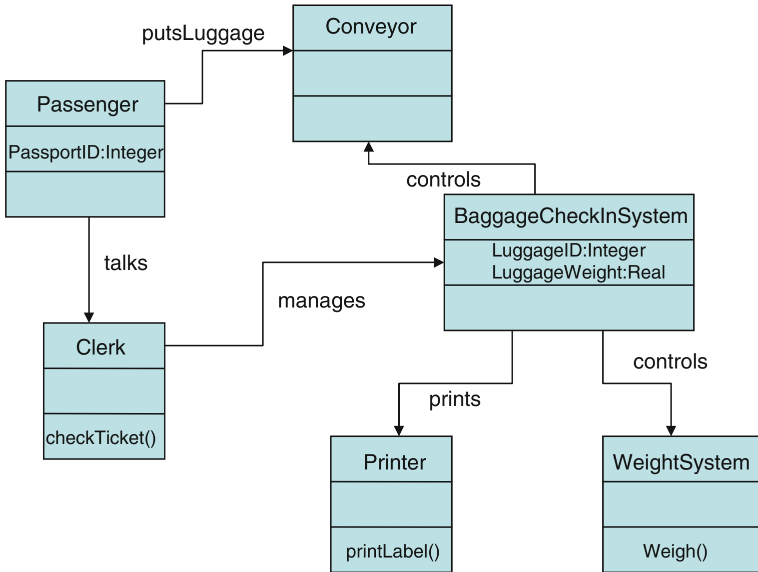

Fig. 3.27 Elaborated class diagram

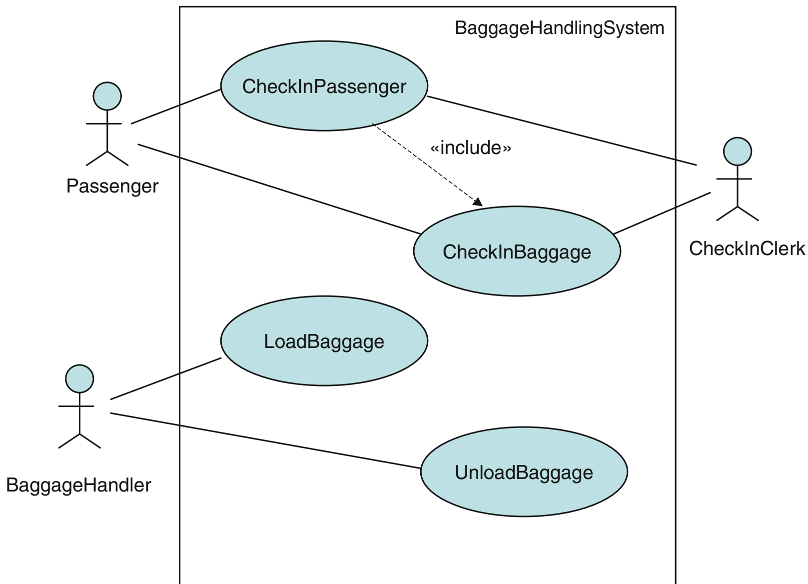

Fig. 3.28 Use case diagram for Baggage Handling System

the diagram and each is attached to a vertical timeline. Messages are ordered by their sequence and are represented by arrows between the timelines. Also included is the feature of an **interaction frame** and the operation 'ref' has been used to

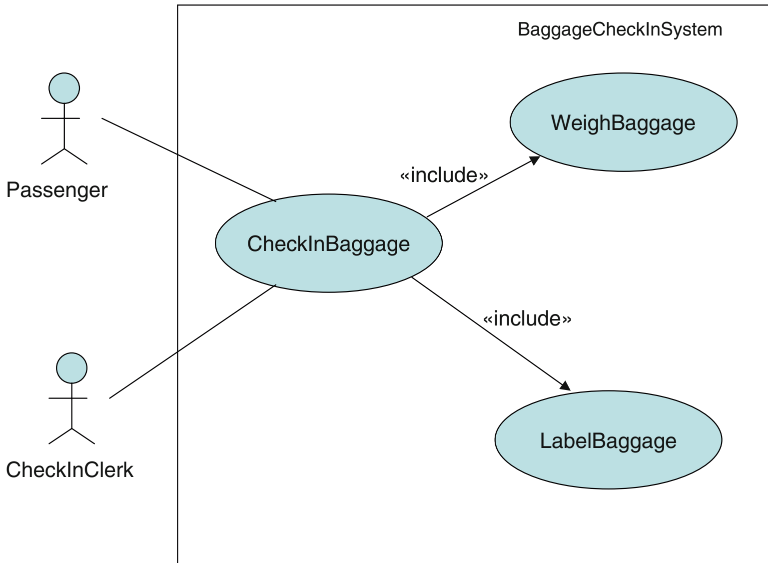

Fig. 3.29 Use case diagram for Baggage Check-in System

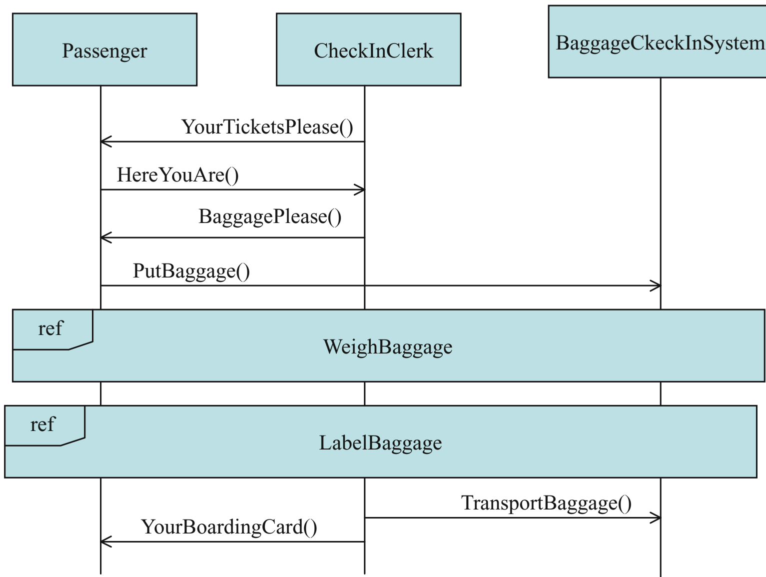

Fig. 3.30 Example sequence diagram

indicate 'reference' i.e. refers to an interaction defined in another diagram, in this case **WeighBaggage** and **LabelBaggage**. These frames have been included to cover the lifelines involved in the interaction.

3.3.4 Formal Methods

Formal methods provide a more rigorous representation based on mathematics, and can be used to conduct mathematical proofs of consistency of specification and correctness of implementation. Rigorous checking is possible, which can eliminate some kinds of errors. This may be necessary in certain types of systems, e.g. nuclear power stations, weapons, and aircraft control systems.

Z (Spivey 1989), VDM (Jones 1986), LOTOS (Bjorner 1987) and the B-Method (Abrial 1996) are the most common formal methods for formal definition of functionality. LOTOS (Language of Temporal Ordering Specification), VDM (the Vienna Definition Language) and Z are formal methods standardized by ISO. B and LOTOS models are executable, and B models can be refined into code.

Formal methods are particularly suitable for critical systems i.e. ones in which potential financial or human loss would be catastrophic, and the cost of applying mathematically rigorous methods can be justified.

Formal methods are slowly becoming more important. If their scope can be broadened to address wider system issues, they will become more useful.

3.3.4.1 Z–A Model-Based Formal Method

Z is a formal specification notation based on first order predicate logic and set theory. The notation allows data to be represented as sets, mappings, tuples, relations, sequences and Cartesian products. There are also functions and operation symbols for manipulating data of these types.

Z specifications are presented in a small, easy to read boxed notation called a 'schema'. Schemas take the form of a signature part and a predicate part. The signature part is a list of variable declarations and the predicate part consists of a single predicate. Naming a schema introduces a syntactic equivalence between the name and the schema. See Fig. 3.31.

Specifications in Z are presented as a collection of schemas where a schema introduces some specification entities and sets out the relationships between them. They provide a framework within which a specification can be developed and presented incrementally.

Figure 3.32 shows a Z specification for the 'issue' operation for a library, where the general behaviour of the overall library system would be specified in a schema

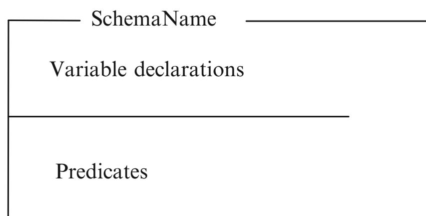

Fig. 3.31 Z schema

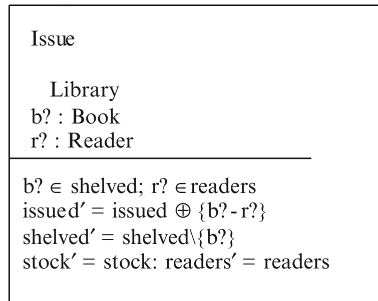

Fig. 3.32 Example schema

named ‘library’. The notation ΔLibrary is called a delta schema and indicates that the ‘Issue’ operation causes a state change to occur in the Library.

The schema in Fig. 3.32 distinguishes between inputs and outputs, and before states and after states. These operations are denoted as follows:

‘?’ denotes the variable as an input to the operation
‘!’ denotes the variable as an output of the operation

A state after the operation is decorated with the “symbol, e.g. stock” to distinguish it from the state before the operation.

3.4 Summary

This chapter has addressed the issues of system modelling, particularly with respect to the solution domain. A range of techniques and methods have been presented ranging from those which have stood the test of time to those which have been developed more recently. All have been widely used in industry. The contents of the chapter provide a basis for the discussion on modelling stakeholder and system requirements in subsequent chapters.