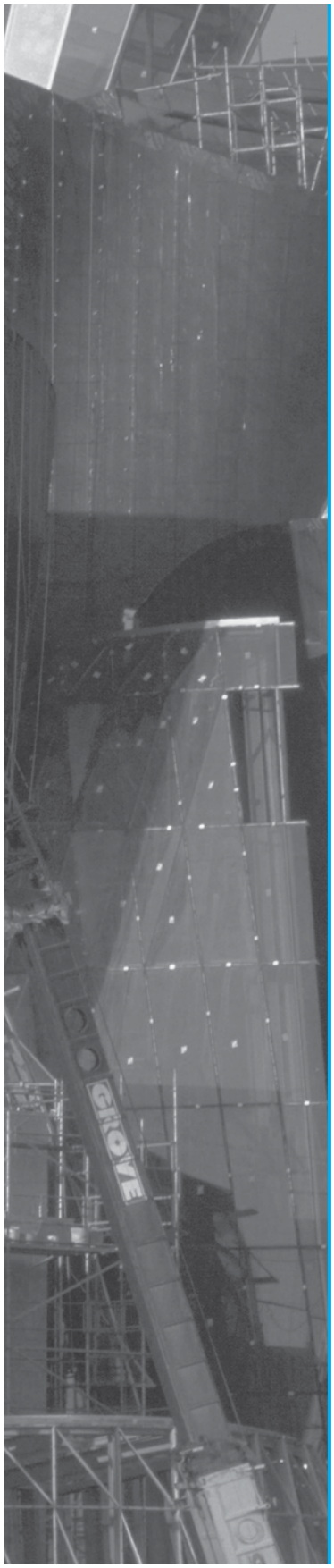

5 System modeling

Objectives

The aim of this chapter is to introduce some types of system model that may be developed as part of the requirements engineering and system design processes. When you have read the chapter, you will:

- understand how graphical models can be used to represent software systems;
- understand why different types of model are required and the fundamental system modeling perspectives of context, interaction, structure, and behavior;
- have been introduced to some of the diagram types in the Unified Modeling Language (UML) and how these diagrams may be used in system modeling;
- be aware of the ideas underlying model-driven engineering, where a system is automatically generated from structural and behavioral models.

Contents

System modeling is the process of developing abstract models of a system, with each model presenting a different view or perspective of that system. System modeling has generally come to mean representing the system using some kind of graphical notation, which is now almost always based on notations in the Unified Modeling Language (UML). However, it is also possible to develop formal (mathematical) models of a system, usually as a detailed system specification. I cover graphical modeling using the UML in this chapter and formal modeling in Chapter 12.

Models are used during the requirements engineering process to help derive the requirements for a system, during the design process to describe the system to engineers implementing the system and after implementation to document the system's structure and operation. You may develop models of both the existing system and the system to be developed:

1. Models of the existing system are used during requirements engineering. They help clarify what the existing system does and can be used as a basis for discussing its strengths and weaknesses. These then lead to requirements for the new system.
2. Models of the new system are used during requirements engineering to help explain the proposed requirements to other system stakeholders. Engineers use these models to discuss design proposals and to document the system for implementation. In a model-driven engineering process, it is possible to generate a complete or partial system implementation from the system model.

The most important aspect of a system model is that it leaves out detail. A 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 very unlikely event of this book being serialized in a newspaper, the presentation there would be an abstraction of the book's key points. If it were 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.

You may develop different models to represent the system from different perspectives. For example:

1. An external perspective, where you model the context or environment of the system.
2. An interaction perspective where you model the interactions between a system and its environment or between the components of a system.
3. A structural perspective, where you model the organization of a system or the structure of the data that is processed by the system.
4. A behavioral perspective, where you model the dynamic behavior of the system and how it responds to events.

These perspectives have much in common with Krutchen's 4 + 1 view of system architecture (Kruchten, 1995), where he suggests that you should document a system's architecture and organization from different perspectives. I discuss this 4 + 1 approach in Chapter 6.

In this chapter, I use diagrams defined in UML (Booch et al., 2005; Rumbaugh et al., 2004), which has become a standard modeling language for object-oriented modeling. The UML has many diagram types and so supports the creation of many different types of system model. However, a survey in 2007 (Erickson and Siau, 2007) showed that most users of the UML thought that five diagram types could represent the essentials of a system:

1. Activity diagrams, which show the activities involved in a process or in data processing.
2. Use case diagrams, which show the interactions between a system and its environment.
3. Sequence diagrams, which show interactions between actors and the system and between system components.
4. Class diagrams, which show the object classes in the system and the associations between these classes.
5. State diagrams, which show how the system reacts to internal and external events.

As I do not have space to discuss all of the UML diagram types here, I focus on how these five key types of diagram are used in system modeling.

When developing system models, you can often be flexible in the way that the graphical notation is used. You do not always need to stick rigidly to the details of a notation. The detail and rigor of a model depends on how you intend to use it. There are three ways in which graphical models are commonly used:

1. As a means of facilitating discussion about an existing or proposed system.
2. As a way of documenting an existing system.
3. As a detailed system description that can be used to generate a system implementation.

In the first case, the purpose of the model is to stimulate the discussion amongst the software engineers involved in developing the system. The models may be incomplete (so long as they cover the key points of the discussion) and they may use the modeling notation informally. This is how models are normally used in so-called 'agile modeling' (Ambler and Jeffries, 2002). When models are used as documentation, they do not have to be complete as you may only wish to develop models for some parts of a system. However, these models have to be correct—they should use the notation correctly and be an accurate description of the system.

The Unified Modeling Language

The Unified Modeling Language is a set of 13 different diagram types that may be used to model software systems. It emerged from work in the 1990s on object-oriented modeling where similar object-oriented notations were integrated to create the UML. A major revision (UML 2) was finalized in 2004. The UML is universally accepted as the standard approach for developing models of software systems. Variants have been proposed for more general system modeling.

http://www.SoftwareEngineering-9.com/Web/UML/

In the third case, where models are used as part of a model-based development process, the system models have to be both complete and correct. The reason for this is that they are used as a basis for generating the source code of the system. Therefore, you have to be very careful not to confuse similar symbols, such as stick and block arrowheads, that have different meanings.

5.1 Context models

At an early stage in the specification of a system, you should decide on the system boundaries. This involves working with system stakeholders to decide what functionality should be included in the system and what is provided by the system's environment. You may decide that automated support for some business processes should be implemented but others should be manual processes or supported by different systems. You should look at possible overlaps in functionality with existing systems and decide where new functionality should be implemented. These decisions should be made early in the process to limit the system costs and the time needed for understanding the system requirements and design.

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 computerized system, the environment of the new system is usually the same as the existing system's environment. In other cases, there is more flexibility, and you decide what constitutes the boundary between the system and its environment during the requirements engineering process.

For example, say you are developing the specification for the patient information system for mental healthcare. This system is intended to manage information about patients attending mental health clinics and the treatments that have been prescribed. In developing the specification for this system, you have to decide whether the system should focus exclusively on collecting information about consultations (using other systems to collect personal information about patients) or whether it should also collect personal patient information. The advantage of relying on other systems for patient information is that you avoid duplicating data. The major disadvantage, however, is that using other systems may make it slower to access information. If these systems are unavailable, then the MHC-PMS cannot be used.

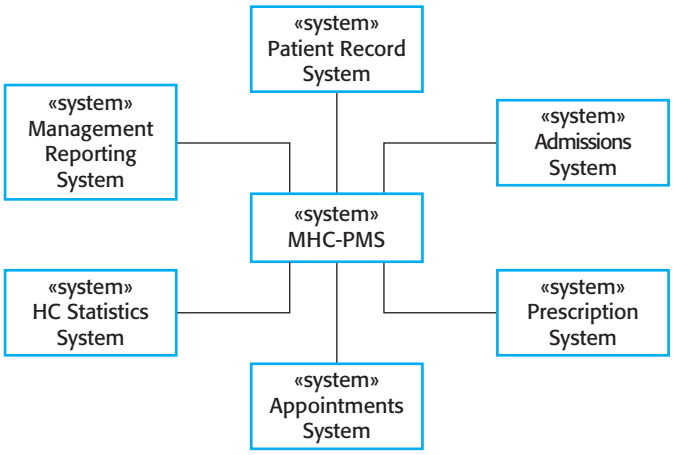

Figure 5.1 The context of the MHC-PMS

The definition of a system boundary is not a value-free judgment. Social and organizational concerns may mean that the position of a system boundary may be determined by non-technical factors. For example, a system boundary may be deliberately 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.

Figure 5.1 is a simple context model that shows the patient information system and the other systems in its environment. From Figure 5.1, you can see that the MHC-PMS is connected to an appointments system and a more general patient record system with which it shares data. The system is also connected to systems for management reporting and hospital bed allocation and a statistics system that collects information for research. Finally, it makes use of a prescription system to generate prescriptions for patients' medication.

Context models normally show that the environment includes several other automated systems. However, they do not show the types of relationships between the 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, or 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 may affect the requirements and design of the system being defined and must be taken into account.

Therefore, simple context models are used along with other models, such as business process models. These describe human and automated processes in which particular software systems are used.

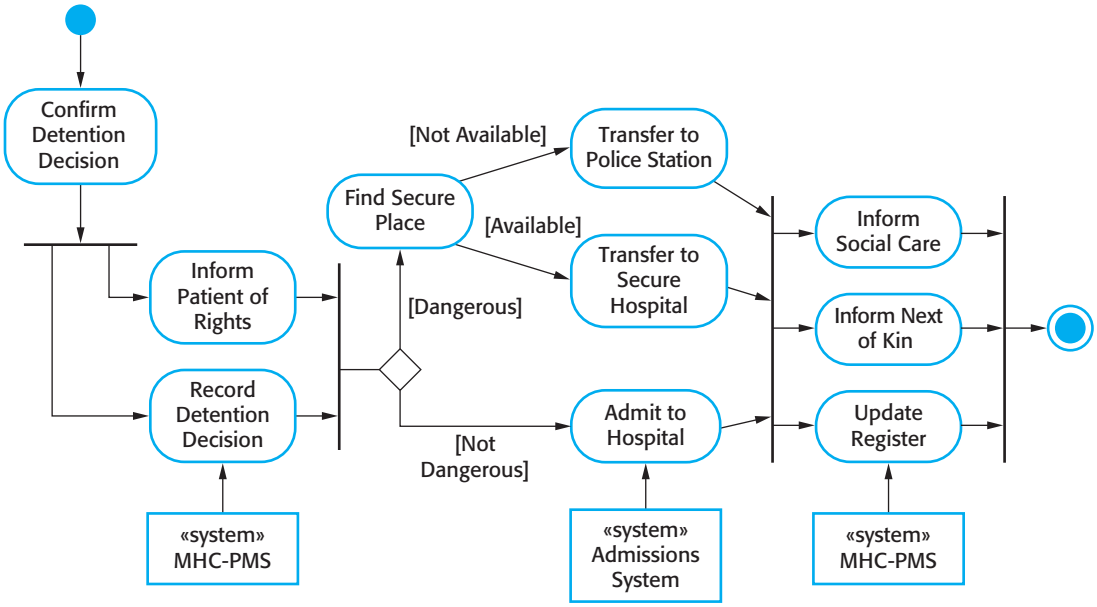

Figure 5.2 Process model of involuntary detention

Figure 5.2 is a model of an important system process that shows the processes in which the MHC-PMS is used. Sometimes, patients who are suffering from mental health problems may be a danger to others or to themselves. They may therefore have to be detained against their will in a hospital so that treatment can be administered. Such detention is subject to strict legal safeguards—for example, the decision to detain a patient must be regularly reviewed so that people are not held indefinitely without good reason. One of the functions of the MHC-PMS is to ensure that such safeguards are implemented.

Figure 5.2 is a UML activity diagram. Activity diagrams are intended to show the activities that make up a system process and the flow of control from one activity to another. The start of a process is indicated by a filled circle; the end by a filled circle inside another circle. Rectangles with round corners represent activities, that is, the specific sub-processes that must be carried out. You may include objects in activity charts. In Figure 5.2, I have shown the systems that are used to support different processes. I have indicated that these are separate systems using the UML stereotype feature.

In a UML activity diagram, arrows represent the flow of work from one activity to another. A solid bar is used to indicate activity coordination. When the flow from more than one activity leads to a solid bar then all of these activities must be complete before progress is possible. When the flow from a solid bar leads to a number of activities, these may be executed in parallel. Therefore, in Figure 5.2, the activities to inform social care and the patient's next of kin, and to update the detention register may be concurrent.

Arrows may be annotated with guards that indicate the condition when that flow is taken. In Figure 5.2, you can see guards showing the flows for patients who are

dangerous and not dangerous to society. Patients who are dangerous to society must be detained in a secure facility. However, patients who are suicidal and so are a danger to themselves may be detained in an appropriate ward in a hospital.

5.2 Interaction models

All systems involve interaction of some kind. This can be user interaction, which involves user inputs and outputs, interaction between the system being developed and other systems or interaction between the components of the system. Modeling user interaction is important as it helps to identify user requirements. Modeling system to system interaction highlights the communication problems that may arise. Modeling component interaction helps us understand if a proposed system structure is likely to deliver the required system performance and dependability.

In this section, I cover two related approaches to interaction modeling:

1. Use case modeling, which is mostly used to model interactions between a system and external actors (users or other systems).
2. Sequence diagrams, which are used to model interactions between system components, although external agents may also be included.

Use case models and sequence diagrams present interaction at different levels of detail and so may be used together. The details of the interactions involved in a high-level use case may be documented in a sequence diagram. The UML also includes communication diagrams that can be used to model interactions. I don't discuss these here as they are an alternative representation of sequence charts. In fact, some tools can generate a communication diagram from a sequence diagram.

5.2.1 Use case modeling

Use case modeling was originally developed by Jacobson et al. (1993) in the 1990s and was incorporated into the first release of the UML (Rumbaugh et al., 1999). As I have discussed in Chapter 4, use case modeling is widely used to support requirements elicitation. A use case can be taken as a simple scenario that describes what a user expects from a system.

Each use case represents a discrete task that involves external interaction with a system. In its simplest form, a use case is shown as an ellipse with the actors involved in the use case represented as stick figures. Figure 5.3 shows a use case from the MHC-PMS that represents the task of uploading data from the MHC-PMS to a more general patient record system. This more general system maintains summary data about a patient rather than the data about each consultation, which is recorded in the MHC-PMS.

Figure 5.3 Transfer-data use case

Notice that there are two actors in this use case: the operator who is transferring the data and the patient record system. The stick figure notation was originally developed to cover human interaction but it is also now used to represent other external systems and hardware. Formally, use case diagrams should use lines without arrows as arrows in the UML indicate the direction of flow of messages. Obviously, in a use case messages pass in both directions. However, the arrows in Figure 5.3 are used informally to indicate that the medical receptionist initiates the transaction and data is transferred to the patient record system.

Use case diagrams give a fairly simple overview of an interaction so you have to provide more detail to understand what is involved. This detail can either be a simple textual description, a structured description in a table, or a sequence diagram as discussed below. You chose the most appropriate format depending on the use case and the level of detail that you think is required in the model. I find a standard tabular format to be the most useful. Figure 5.4 shows a tabular description of the 'Transfer data' use case.

As I have discussed in Chapter 4, composite use case diagrams show a number of different use cases. Sometimes, it is possible to include all possible interactions with a system in a single composite use case diagram. However, this may be impossible because of the number of use cases. In such cases, you may develop several diagrams, each of which shows related use cases. For example, Figure 5.5 shows all of the use cases in the MHC-PMS in which the actor 'Medical Receptionist' is involved.

Figure 5.4 Tabular description of the 'Transfer data' use case

MHC-PMS: Transfer data	
Actors	Medical receptionist, patient records system (PRS)
Description	A receptionist may transfer data from the MHC-PMS to a general patient record database that is maintained by a health authority. The information transferred may either be updated personal information (address, phone number, etc.) or a summary of the patient's diagnosis and treatment.
Data	Patient's personal information, treatment summary
Stimulus	User command issued by medical receptionist
Response	Confirmation that PRS has been updated
Comments	The receptionist must have appropriate security permissions to access the patient information and the PRS.

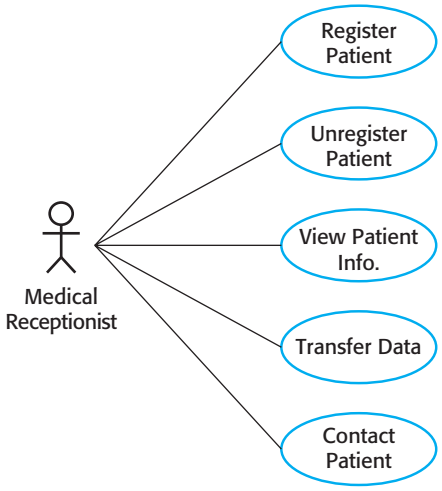

Figure 5.5 Use cases involving the role 'medical receptionist'

5.2.2 Sequence diagrams

Sequence diagrams in the UML are primarily used to model the interactions between the actors and the objects in a system and the interactions between the objects themselves. The UML has a rich syntax for sequence diagrams, which allows many different kinds of interaction to be modeled. I don't have space to cover all possibilities here so I focus on the basics of this diagram type.

As the name implies, a sequence diagram shows the sequence of interactions that take place during a particular use case or use case instance. Figure 5.6 is an example of a sequence diagram that illustrates the basics of the notation. This diagram models the interactions involved in the View patient information use case, where a medical receptionist can see some patient information.

The objects and actors involved are listed along the top of the diagram, with a dotted line drawn vertically from these. Interactions between objects are indicated by annotated arrows. The rectangle on the dotted lines indicates the lifeline of the object concerned (i.e., the time that object instance is involved in the computation). You read the sequence of interactions from top to bottom. The annotations on the arrows indicate the calls to the objects, their parameters, and the return values. In this example, I also show the notation used to denote alternatives. A box named alt is used with the conditions indicated in square brackets.

You can read Figure 5.6 as follows:

1. The medical receptionist triggers the ViewInfo method in an instance P of the PatientInfo object class, supplying the patient's identifier, PID. P is a user interface object, which is displayed as a form showing patient information.
2. The instance P calls the database to return the information required, supplying the receptionist's identifier to allow security checking (at this stage, we do not care where this UID comes from).

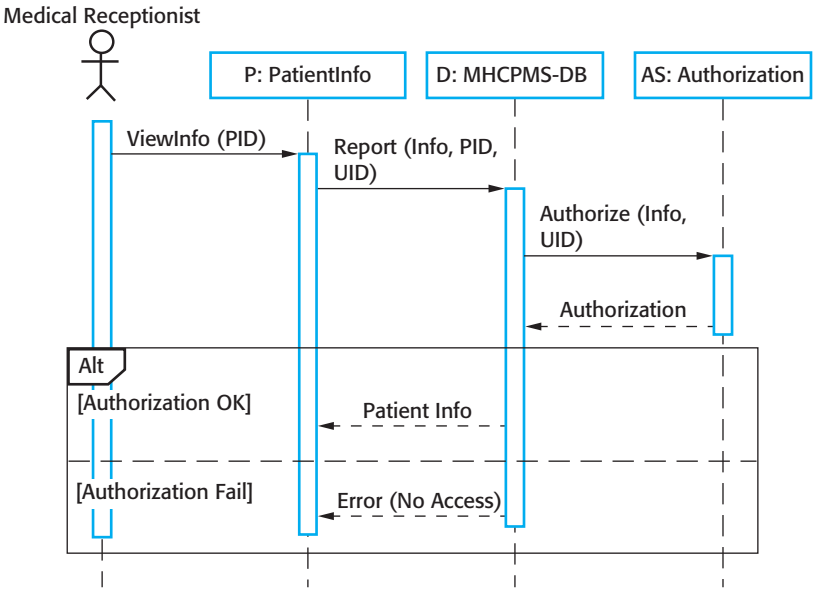

Figure 5.6 Sequence diagram for View patient information

3. The database checks with an authorization system that the user is authorized for this action.
4. If authorized, the patient information is returned and a form on the user's screen is filled in. If authorization fails, then an error message is returned.

Figure 5.7 is a second example of a sequence diagram from the same system that illustrates two additional features. These are the direct communication between the actors in the system and the creation of objects as part of a sequence of operations. In this example, an object of type Summary is created to hold the summary data that is to be uploaded to the PRS (patient record system). You can read this diagram as follows:

1. The receptionist logs on to the PRS.
2. There are two options available. These allow the direct transfer of updated patient information to the PRS and the transfer of summary health data from the MHC-PMS to the PRS.
3. In each case, the receptionist's permissions are checked using the authorization system.
4. Personal information may be transferred directly from the user interface object to the PRS. Alternatively, a summary record may be created from the database and that record is then transferred.
5. On completion of the transfer, the PRS issues a status message and the user logs off.

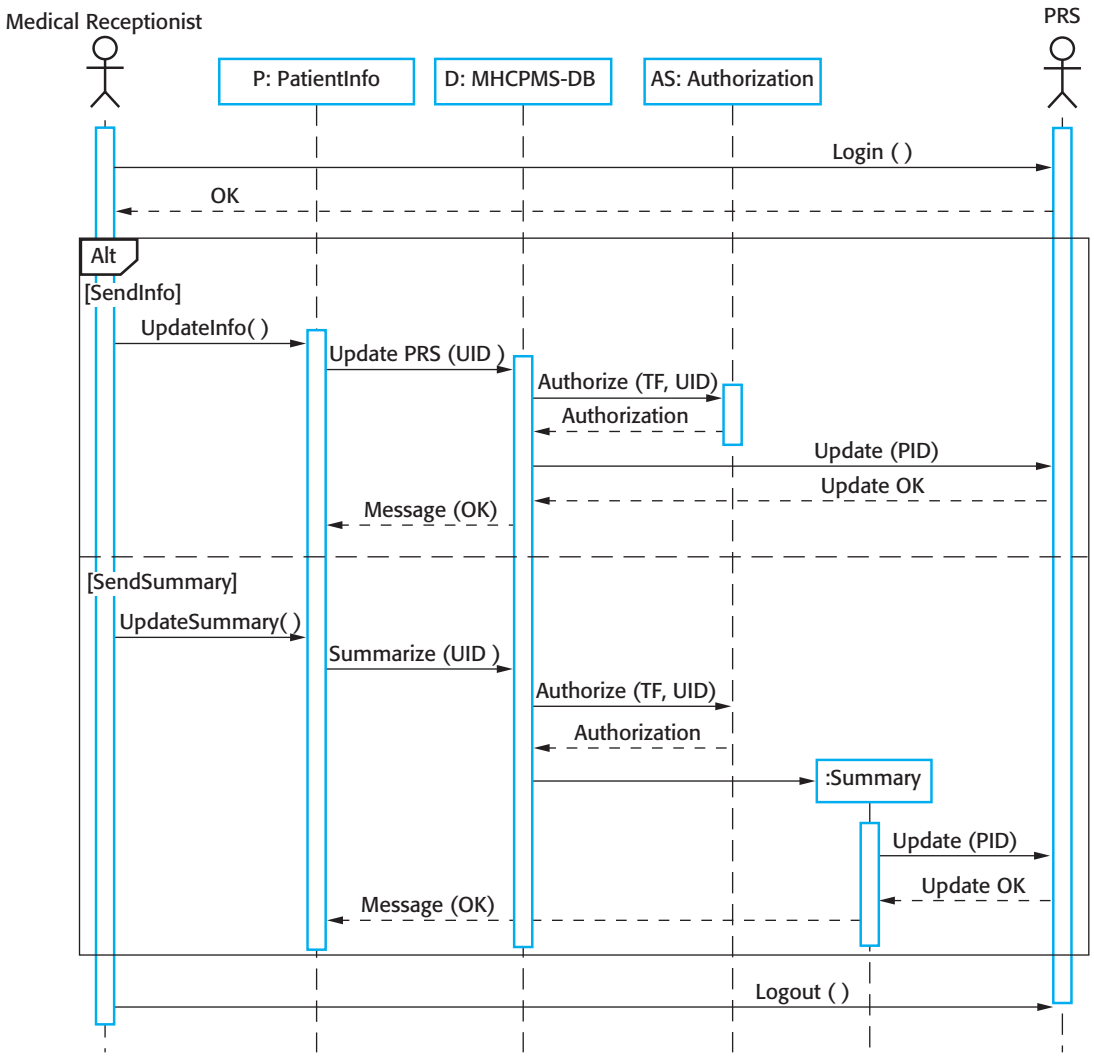

Figure 5.7 Sequence diagram for transfer data

Unless you are using sequence diagrams for code generation or detailed documentation, you don't have to include every interaction in these diagrams. If you develop system models early in the development process to support requirements engineering and high-level design, there will be many interactions which depend on implementation decisions. For example, in Figure 5.7 the decision on how to get the user's identifier to check authorization is one that can be delayed. In an implementation, this might involve interacting with a User object but this is not important at this stage and so need not be included in the sequence diagram.

Object-oriented requirements analysis

In object-oriented requirements analysis, you model real-world entities using object classes. You may create different types of object model, 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. These each present unique information about the system that is being specified.

http://www.SoftwareEngineering-9.com/Web/OORA/

5.3 Structural models

Structural models of software display the organization of a system in terms of the components that make up that system and their relationships. Structural models may be static models, which show the structure of the system design or dynamic models, which show the organization of the system when it is executing. These are not the same things—the dynamic organization of a system as a set of interacting threads may be very different from a static model of the system components.

You create structural models of a system when you are discussing and designing the system architecture. Architectural design is a particularly important topic in software engineering and UML component, package, and deployment diagrams may all be used when presenting architectural models. I cover different aspects of software architecture and architectural modeling in Chapters 6, 18, and 19. In this section, I focus on the use of class diagrams for modeling the static structure of the object classes in a software system.

5.3.1 Class diagrams

Class diagrams are used when developing an object-oriented system model to show the classes in a system and the associations between these classes. Loosely, an object class can be thought of as a general definition of one kind of system object. An association is a link between classes that indicates that there is a relationship between these classes. Consequently, each class may have to have some knowledge of its associated class.

When you are developing models during the early stages of the software engineering process, objects represent something in the real world, such as a patient, a prescription, a doctor, etc. As an implementation is developed, you usually need to define additional implementation objects that are used to provide the required system functionality. Here, I focus on the modeling of real-world objects as part of the requirements or early software design processes.

Class diagrams in the UML can be expressed at different levels of detail. When you are developing a model, the first stage is usually to look at the world, identify the essential objects, and represent these as classes. The simplest way of writing these is to write the class name in a box. You can also simply note the existence of an association

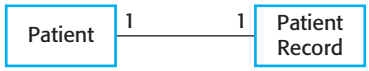

Figure 5.8 UML classes and association

by drawing a line between classes. For example, Figure 5.8 is a simple class diagram showing two classes: Patient and Patient Record with an association between them.

In Figure 5.8, I illustrate a further feature of class diagrams—the ability to show how many objects are involved in the association. In this example, each end of the association is annotated with a 1, meaning that there is a 1:1 relationship between objects of these classes. That is, each patient has exactly one record and each record maintains information about exactly one patient. As you can see from later examples, other multiplicities are possible. You can define that an exact number of objects are involved or, by using a *, as shown in Figure 5.9, that there are an indefinite number of objects involved in the association.

Figure 5.9 develops this type of class diagram to show that objects of class Patient are also involved in relationships with a number of other classes. In this example, I show that you can name associations to give the reader an indication of the type of relationship that exists. The UML also allows the role of the objects participating in the association to be specified.

At this level of detail, class diagrams look like semantic data models. Semantic data models are used in database design. They show the data entities, their associated attributes, and the relations between these entities. This approach to modeling was first proposed in the mid-1970s by Chen (1976); several variants have been developed since then (Codd, 1979; Hammer and McLeod, 1981; Hull and King, 1987), all with the same basic form.

The UML does not include a specific notation for this database modeling as it assumes an object-oriented development process and models data using objects and their relationships. However, you can use the UML to represent a semantic data model. You can think of entities in a semantic data model as simplified object classes

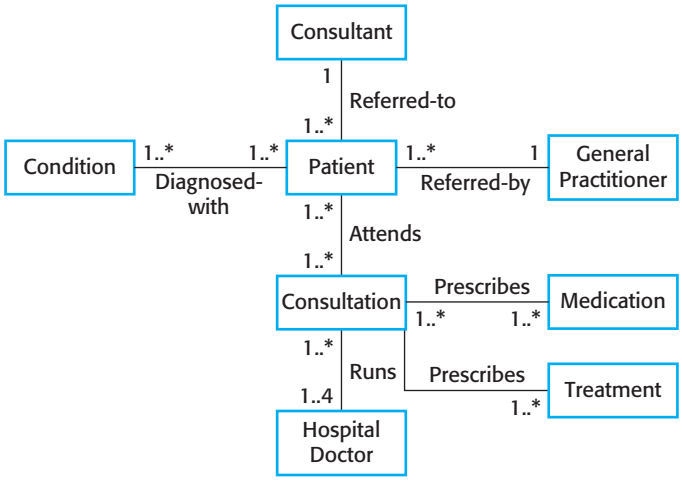

Figure 5.9 Classes and associations in the MHC-PMS

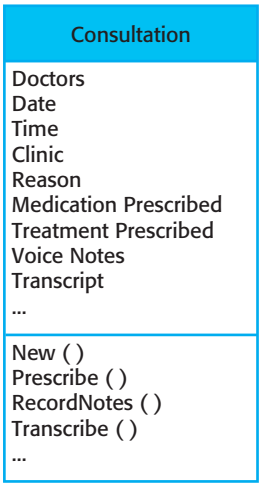

Figure 5.10 The consultation class

(they have no operations), attributes as object class attributes and relations as named associations between object classes.

When showing the associations between classes, it is convenient to represent these classes in the simplest possible way. To define them in more detail, you add information about their attributes (the characteristics of an object) and operations (the things that you can request from an object). For example, a Patient object will have the attribute Address and you may include an operation called ChangeAddress, which is called when a patient indicates that they have moved from one address to another. In the UML, you show attributes and operations by extending the simple rectangle that represents a class. This is illustrated in Figure 5.10 where:

1. The name of the object class is in the top section.
2. The class attributes are in the middle section. This must include the attribute names and, optionally, their types.
3. The operations (called methods in Java and other OO programming languages) associated with the object class are in the lower section of the rectangle.

Figure 5.10 shows possible attributes and operations on the class Consultation. In this example, I assume that doctors record voice notes that are transcribed later to record details of the consultation. To prescribe medication, the doctor involved must use the Prescribe method to generate an electronic prescription.

5.3.2 Generalization

Generalization is an everyday technique that we use to manage complexity. Rather than learn the detailed characteristics of every entity that we experience, we place these entities in more general classes (animals, cars, houses, etc.) and learn the characteristics of

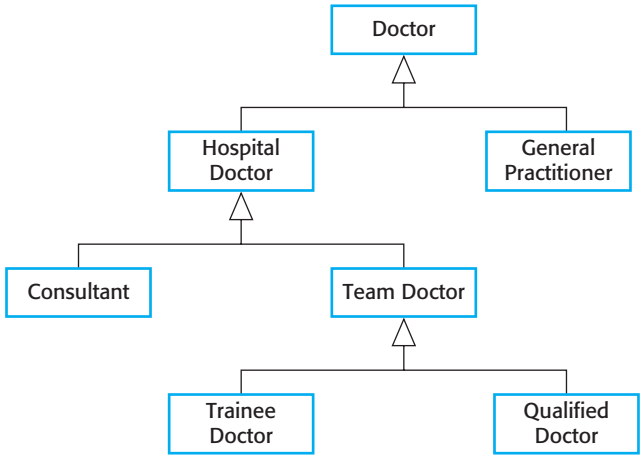

Figure 5.11 A generalization hierarchy

these classes. This allows us to infer that different members of these classes have some common characteristics (e.g., squirrels and rats are rodents). We can make general statements that apply to all class members (e.g., all rodents have teeth for gnawing).

In modeling systems, it is often useful to examine the classes in a system to see if there is scope for generalization. This means that common information will be maintained in one place only. This is good design practice as it means that, if changes are proposed, then you do not have to look at all classes in the system to see if they are affected by the change. In object-oriented languages, such as Java, generalization is implemented using the class inheritance mechanisms built into the language.

The UML has a specific type of association to denote generalization, as illustrated in Figure 5.11. The generalization is shown as an arrowhead pointing up to the more general class. This shows that general practitioners and hospital doctors can be generalized as doctors and that there are three types of Hospital Doctor—those that have just graduated from medical school and have to be supervised (Trainee Doctor); those that can work unsupervised as part of a consultant's team (Registered Doctor); and consultants, who are senior doctors with full decision-making responsibilities.

In a generalization, the attributes and operations associated with higher-level classes are also associated with the lower-level classes. In essence, the lower-level classes are subclasses inherit the attributes and operations from their superclasses. These lower-level classes then add more specific attributes and operations. For example, all doctors have a name and phone number; all hospital doctors have a staff number and a department but general practitioners don't have these attributes as they work independently. They do however, have a practice name and address. This is illustrated in Figure 5.12, which shows part of the generalization hierarchy that I have extended with class attributes. The operations associated with the class Doctor are intended to register and de-register that doctor with the MHC-PMS.

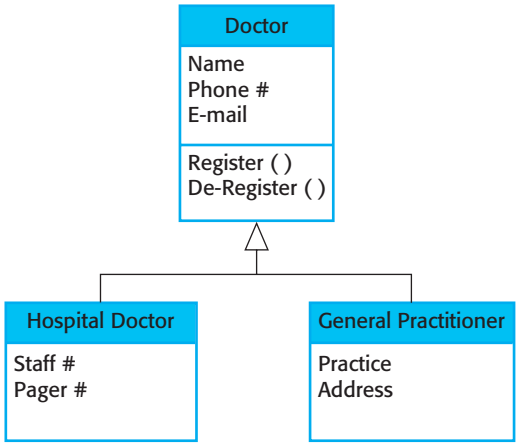

Figure 5.12 A generalization hierarchy with added detail

5.3.3 Aggregation

Objects in the real world are often composed of different parts. For example, a study pack for a course may be composed of a book, PowerPoint slides, quizzes, and recommendations for further reading. Sometimes in a system model, you need to illustrate this. The UML provides a special type of association between classes called aggregation that means that one object (the whole) is composed of other objects (the parts). To show this, we use a diamond shape next to the class that represents the whole. This is shown in Figure 5.13, which shows that a patient record is a composition of Patient and an indefinite number of Consultations.

5.4 Behavioral models

Behavioral models are models of the dynamic behavior of the system as it is executing. They show what happens or what is supposed to happen when a system responds to a stimulus from its environment. You can think of these stimuli as being of two types:

1. *Data* Some data arrives that has to be processed by the system.
2. *Events* Some event happens that triggers system processing. Events may have associated data but this is not always the case.

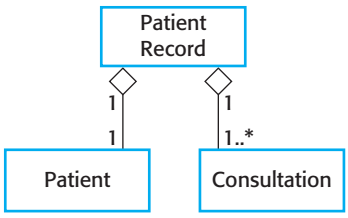

Figure 5.13 The aggregation association

Data-flow diagrams

Data-flow diagrams (DFDs) are system models that show a functional perspective where each transformation represents a single function or process. DFDs are used to show how data flows through a sequence of processing steps. For example, a processing step could be the filtering of duplicate records in a customer database. The data is transformed at each step before moving on to the next stage. These processing steps or transformations represent software processes or functions where data-flow diagrams are used to document a software design.

http://www.SoftwareEngineering-9.com/Web/DFDs

Many business systems are data processing systems that are primarily driven by data. They are controlled by the data input to the system with relatively little external event processing. Their processing involves a sequence of actions on that data and the generation of an output. For example, a phone billing system will accept information about calls made by a customer, calculate the costs of these calls, and generate a bill to be sent to that customer. By contrast, real-time systems are often event driven with minimal data processing. For example, a landline phone switching system responds to events such as 'receiver off hook' by generating a dial tone, or the pressing of keys on a handset by capturing the phone number, etc.

5.4.1 Data-driven modeling

Data-driven models show the sequence of actions involved in processing input data and generating an associated output. They are particularly useful during the analysis of requirements as they can be used to show end-to-end processing in a system. That is, they show the entire sequence of actions that take place from an input being processed to the corresponding output, which is the system's response.

Data-driven models were amongst the first graphical software models. In the 1970s, structured methods such as DeMarco's Structured Analysis (DeMarco, 1978) introduced data-flow diagrams (DFDs) as a way of illustrating the processing steps in a system. Data-flow models are useful because tracking and documenting how the data associated with a particular process moves through the system helps analysts and designers understand what is going on. Data-flow diagrams are simple and intuitive and it is usually possible to explain them to potential system users who can then participate in validating the model.

The UML does not support data-flow diagrams as they were originally proposed and used for modeling data processing. The reason for this is that DFDs focus on system functions and do not recognize system objects. However, because data-driven systems are so common in business, UML 2.0 introduced activity diagrams, which are similar to data-flow diagrams. For example, Figure 5.14 shows the chain of processing involved in the insulin pump software. In this diagram, you can see the processing steps (represented as activities) and the data flowing between these steps (represented as objects).

An alternative way of showing the sequence of processing in a system is to use UML sequence diagrams. You have seen how these can be used to model interaction but, if

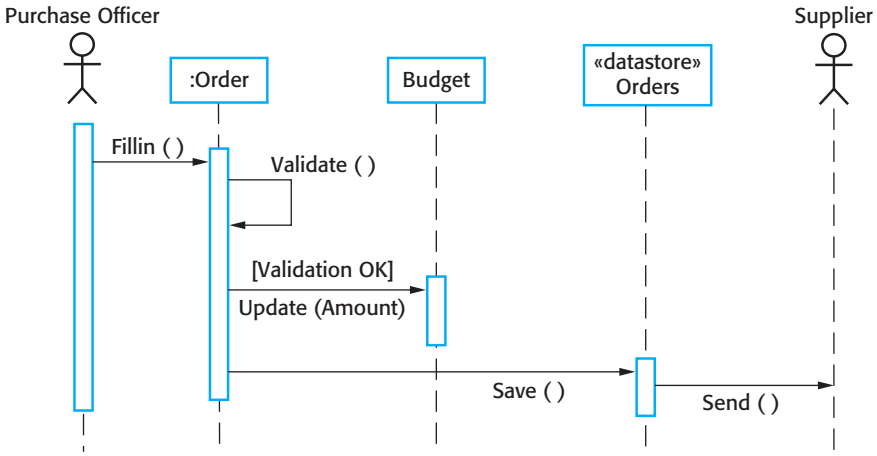

Figure 5.14 An activity model of the insulin pump's operation

you draw these so that messages are only sent from left to right, then they show the sequential data processing in the system. Figure 5.15 illustrates this, using a sequence model of the processing of an order and sending it to a supplier. Sequence models highlight objects in a system, whereas data-flow diagrams highlight the functions. The equivalent data-flow diagram for order processing is shown on the book's web pages.

5.4.2 Event-driven modeling

Event-driven modeling shows how a system responds to external and internal events. It is based on the assumption that a system has a finite number of states and that events (stimuli) may cause a transition from one state to another. 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. This view of a system is particularly appropriate for real-time systems. Event-based modeling was introduced in real-time design methods such as those proposed by Ward and Mellor (1985) and Harel (1987, 1988).

The UML supports event-based modeling using state diagrams, which were based on Statecharts (Harel, 1987, 1988). State diagrams show system states and events that cause

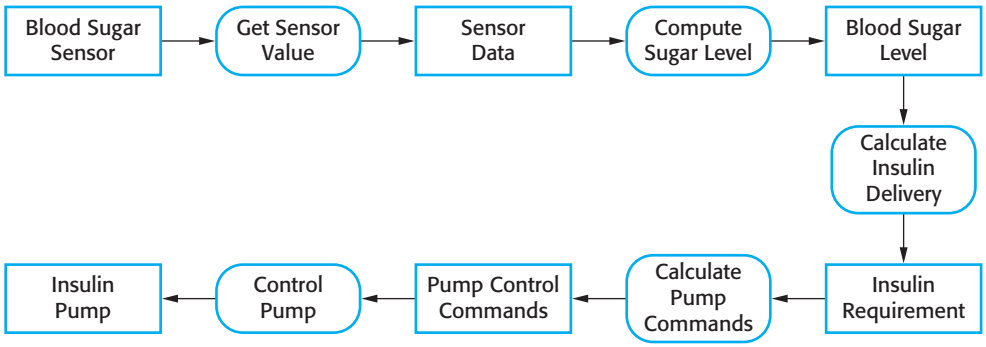

Figure 5.15 Order processing

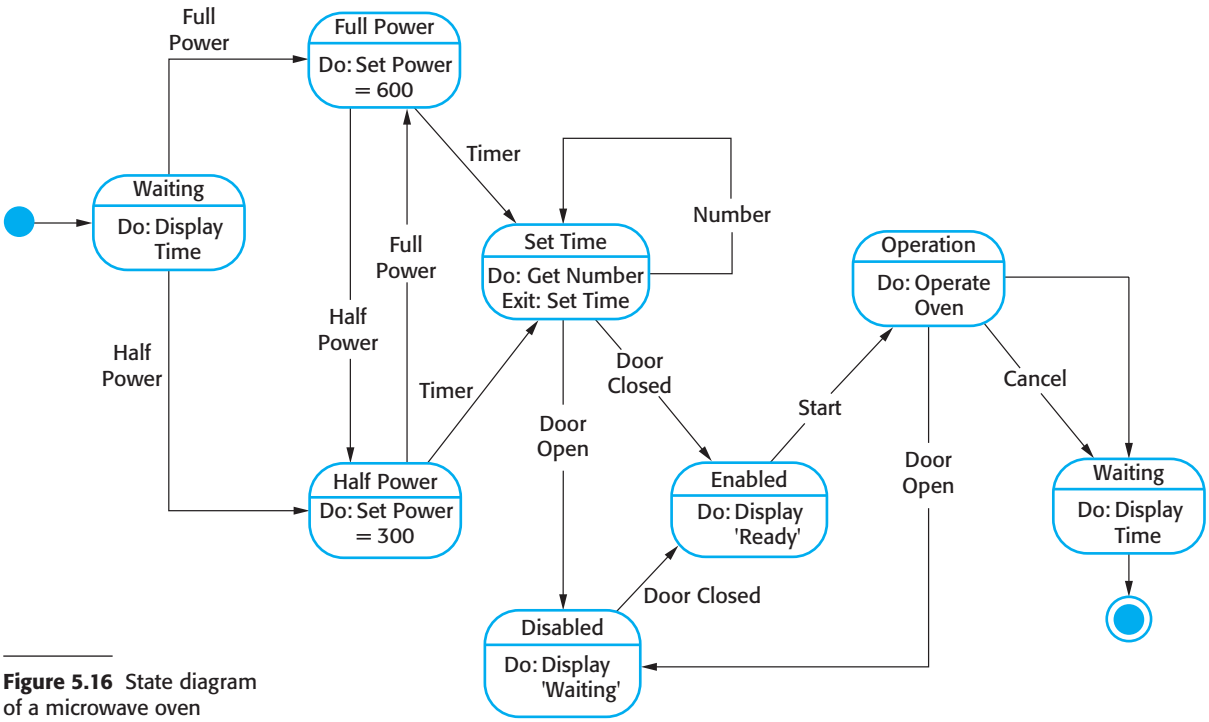

Figure 5.16 State diagram of a microwave oven

transitions from one state to another. They do not show the flow of data within the system but may include additional information on the computations carried out in each state.

I use an example of control software for a very simple microwave oven to illustrate event-driven modeling. Real microwave ovens are actually much more complex than this system but the simplified system is easier to understand. This simple microwave has a switch to select full or half power, a numeric keypad to input the cooking time, a start/stop button, and an alphanumeric display.

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 using a numeric keypad.
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 that is used to display various alerts and warning messages.

In UML state diagrams, rounded rectangles represent system states. They may include a brief description (following 'do') of the actions taken in that state. The labeled arrows represent stimuli that force a transition from one state to another. You can indicate start and end states using filled circles, as in activity diagrams.

From Figure 5.16, you can see that the system starts in a waiting state and responds initially to either the full-power or the half-power button. Users can change

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 five 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.

Figure 5.17 States and stimuli for the microwave oven

their mind after selecting one of these and press the other button. The time is set and, if the door is closed, the Start button is enabled. Pushing this button starts the oven operation and cooking takes place for the specified time. This is the end of the cooking cycle and the system returns to the waiting state.

The UML notation lets you indicate the activity that takes place in a state. In a detailed system specification you have to provide more detail about both the stimuli and the system states. I illustrate this in Figure 5.17, which shows a tabular description of each state and how the stimuli that force state transitions are generated.

The problem with state-based modeling is that the number of possible states increases rapidly. For large system models, therefore, you need to hide detail in the

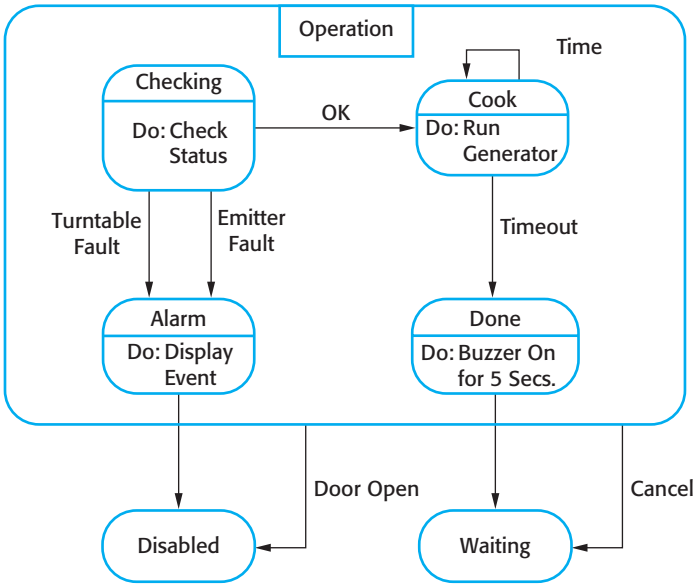

Figure 5.18 Microwave oven operation

models. One way to do this is by using the notion of a superstate that encapsulates a number of separate states. This superstate looks like a single state on a high-level model but is then expanded to show more detail on a separate diagram. To illustrate this concept, consider the Operation state in Figure 5.15. This is a superstate that can be expanded, as illustrated in Figure 5.18.

The Operation state includes a number of sub-states. It shows that operation starts with a status check and that if any problems are discovered an alarm is indicated and operation is disabled. Cooking involves running the microwave generator for the specified time; on completion, a buzzer is sounded. If the door is opened during operation, the system moves to the disabled state, as shown in Figure 5.15.

5.5 Model-driven engineering

Model-driven engineering (MDE) is an approach to software development where models rather than programs are the principal outputs of the development process (Kent, 2002; Schmidt, 2006). The programs that execute on a hardware/software platform are then generated automatically from the models. Proponents of MDE argue that this raises the level of abstraction in software engineering so that engineers no longer have to be concerned with programming language details or the specifics of execution platforms.

Model-driven engineering has its roots in model-driven architecture (MDA) which was proposed by the Object Management Group (OMG) in 2001 as a new software development paradigm. Model-driven engineering and model-driven architecture are often seen as the same thing. However, I think that MDE has a wider scope than

MDA. As I discuss later in this section, MDA focuses on the design and implementation stages of software development whereas MDE is concerned with all aspects of the software engineering process. Therefore, topics such as model-based requirements engineering, software processes for model-based development, and model-based testing are part of MDE but not, currently, part of MDA.

Although MDA has been in use since 2001, model-based engineering is still at an early stage of development and it is unclear whether or not it will have a significant effect on software engineering practice. The main arguments for and against MDE are:

1. *For MDE* Model-based engineering allows engineers to think about systems at a high level of abstraction, without concern for the details of their implementation. This reduces the likelihood of errors, speeds up the design and implementation process, and allows for the creation of reusable, platform-independent application models. By using powerful tools, system implementations can be generated for different platforms from the same model. Therefore, to adapt the system to some new platform technology, it is only necessary to write a translator for that platform. When this is available, all platform-independent models can be rapidly rehosted on the new platform.

2. *Against MDE* As I discussed earlier in this chapter, models are a good way of facilitating discussions about a software design. However, it does not always follow that the abstractions that are supported by the model are the right abstractions for implementation. So, you may create informal design models but then go on to implement the system using an off-the-shelf, configurable package. Furthermore, the arguments for platform independence are only valid for large long-lifetime systems where the platforms become obsolete during a system's lifetime. However, for this class of systems, we know that implementation is not the major problem—requirements engineering, security and dependability, integration with legacy systems, and testing are more significant.

There have been significant MDE success stories reported by the OMG on their Web pages (www.omg.org/mda/products_success.htm) and the approach is used within large companies such as IBM and Siemens. The techniques have been used successfully in the development of large, long-lifetime software systems such as air traffic management systems. Nevertheless, at the time of writing, model-driven approaches are not widely used for software engineering. Like formal methods of software engineering, which I discuss in Chapter 12, I believe that MDE is an important development. However, as is also the case with formal methods, it is not clear whether the costs and risks of model-driven approaches outweigh the possible benefits.

5.5.1 Model-driven architecture

Model-driven architecture (Kleppe, et al., 2003; Mellor et al., 2004; Stahl and Voelter, 2006) is a model-focused approach to software design and implementation that uses a sub-set of UML models to describe a system. Here, models at different

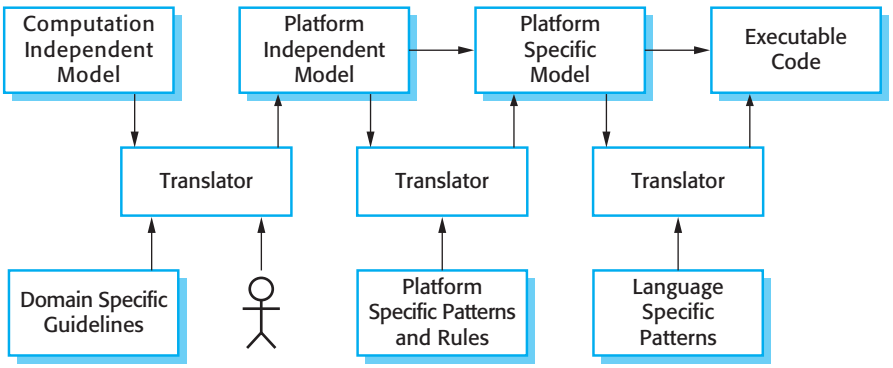

Figure 5.19 MDA transformations

levels of abstraction are created. From a high-level platform independent model it is possible, in principle, to generate a working program without manual intervention.

The MDA method recommends that three types of abstract system model should be produced:

1. A computation independent model (CIM) that models the important domain abstractions used in the system. CIMs are sometimes called domain models. You may develop several different CIMs, reflecting different views of the system. For example, there may be a security CIM in which you identify important security abstractions such as an asset and a role and a patient record CIM, in which you describe abstractions such as patients, consultations, etc.
2. A platform independent model (PIM) that models the operation of the system without reference to its implementation. The PIM is usually described using UML models that show the static system structure and how it responds to external and internal events.
3. Platform specific models (PSM) which are transformations of the platform-independent model with a separate PSM for each application platform. In principle, there may be layers of PSM, with each layer adding some platform-specific detail. So, the first-level PSM could be middleware-specific but database independent. When a specific database has been chosen, a database-specific PSM can then be generated.

As I have said, transformations between these models may be defined and applied automatically by software tools. This is illustrated in Figure 5.19, which also shows a final level of automatic transformation. A transformation is applied to the PSM to generate executable code that runs on the designated software platform.

At the time of writing, automatic CIM to PIM translation is still at the research prototype stage. It is unlikely that completely automated translation tools will be available in the near future. Human intervention, indicated by a stick figure in Figure 5.19, will be needed for the foreseeable future. CIMs are related and part of the translation

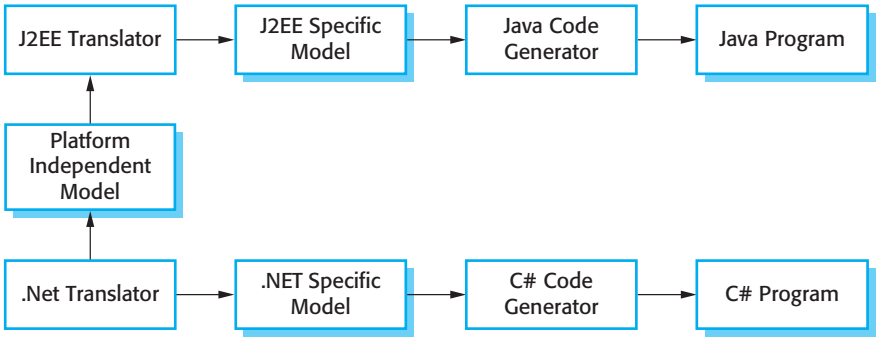

Figure 5.20 Multiple platform-specific models

process may involve linking concepts in different CIMs. For example, the concept of a role in a security CIM may be mapped onto the concept of a staff member in a hospital CIM. Mellor and Balcer (2002) give the name 'bridges' to the information that supports mapping from one CIM to another.

The translation of PIMs to PSMs is more mature and several commercial tools are available that provide translators from PIMs to common platforms such as Java and J2EE. These rely on an extensive library of platform-specific rules and patterns to convert the PIM to the PSM. There may be several PSMs for each PIM in the system. If a software system is intended to run on different platforms (e.g., J2EE and .NET), then it is only necessary to maintain the PIM. The PSMs for each platform are automatically generated. This is illustrated in Figure 5.20.

Although MDA-support tools include platform-specific translators, it is often the case that these will only offer partial support for the translation from PIMs to PSMs. In the vast majority of cases, the execution environment for a system is more than the standard execution platform (e.g., J2EE, .NET, etc.). It also includes other application systems, application libraries that are specific to a company, and user interface libraries. As these vary significantly from one company to another, standard tool support is not available. Therefore, when MDA is introduced, special purpose translators may have to be created that take the characteristics of the local environment into account. In some cases (e.g., for user interface generation), completely automated PIM to PSM translation may be impossible.

There is an uneasy relationship between agile methods and model-driven architecture. The notion of extensive up-front modeling contradicts the fundamental ideas in the agile manifesto and I suspect that few agile developers feel comfortable with model-driven engineering. The developers of MDA claim that it is intended to support an iterative approach to development and so can be used within agile methods (Mellor, et al., 2004). If transformations can be completely automated and a complete program generated from a PIM, then, in principle, MDA could be used in an agile development process as no separate coding would be required. However, as far as I am aware, there are no MDA tools that support practices such as regression testing and test-driven development.

5.5.2 Executable UML

The fundamental notion behind model-driven engineering is that completely automated transformation of models to code should be possible. To achieve this, you have to be able to construct graphical models whose semantics are well defined. You also need a way of adding information to graphical models about the ways in which the operations defined in the model are implemented. This is possible using a subset of UML 2, called Executable UML or xUML (Mellor and Balcer, 2002). I don't have space here to describe the details of xUML, so I simply present a short overview of its main features.

UML was designed as a language for supporting and documenting software design, not as a programming language. The designers of UML were not concerned with semantic details of the language but with its expressiveness. They introduced useful notions such as use case diagrams that help with the design but which are too informal to support execution. To create an executable sub-set of UML, the number of model types has therefore been dramatically reduced to three key model types:

1. Domain models identify the principal concerns in the system. These are defined using UML class diagrams that include objects, attributes, and associations.
2. Class models, in which classes are defined, along with their attributes and operations.
3. State models, in which a state diagram is associated with each class and is used to describe the lifecycle of the class.

The dynamic behavior of the system may be specified declaratively using the object constraint language (OCL) or may be expressed using UML's action language. The action language is like a very high-level programming language where you can refer to objects and their attributes and specify actions to be carried out.

KEY POINTS

- A model is an abstract view of a system that ignores some system details. Complementary system models can be developed to show the system's context, interactions, structure, and behavior.
- Context models show how a system that is being modeled is positioned in an environment with other systems and processes. They help define the boundaries of the system to be developed.
- Use case diagrams and sequence diagrams are used to describe the interactions between user the system being designed and users/other systems. Use cases describe interactions between a system and external actors; sequence diagrams add more information to these by showing interactions between system objects.

- Structural models show the organization and architecture of a system. Class diagrams are used to define the static structure of classes in a system and their associations.
- Behavioral models are used to describe the dynamic behavior of an executing system. This can be modeled from the perspective of the data processed by the system or by the events that stimulate responses from a system.
- Activity diagrams may be used to model the processing of data, where each activity represents one process step.
- State diagrams are used to model a system's behavior in response to internal or external events.
- Model-driven engineering is an approach to software development in which a system is represented as a set of models that can be automatically transformed to executable code.

FURTHER READING

Requirements Analysis and System Design. This book focuses on information systems analysis and discusses how different UML models can be used in the analysis process. (L. Maciaszek, Addison-Wesley, 2001.)

MDA Distilled: Principles of Model-driven Architecture .This is a concise and accessible introduction to the MDA method. It is written by enthusiasts so the book says very little about possible problems with this approach. (S. J. Mellor, K. Scott and D. Weise, Addison-Wesley, 2004.)

Using UML: Software Engineering with Objects and Components, 2nd ed. A short, readable introduction to the use of the UML in system specification and design. This book is excellent for learning and understanding the UML, although it is not a full description of the notation. (P. Stevens with R. Pooley, Addison-Wesley, 2006.)

EXERCISES

5.1. Explain why it is important to model the context of a system that is being developed. Give two examples of possible errors that could arise if software engineers do not understand the system context.

5.2. How might you use a model of a system that already exists? Explain why it is not always necessary for such a system model to be complete and correct. Would the same be true if you were developing a model of a new system?

5.3. You have been asked to develop a system that will help with planning large-scale events and parties such as weddings, graduation celebrations, birthday parties, etc. Using an activity diagram, model the process context for such a system that shows the activities involved in planning a party (booking a venue, organizing invitations, etc.) and the system elements that may be used at each stage.

5.4. For the MHC-PMS, propose a set of use cases that illustrates the interactions between a doctor, who sees patients and prescribes medicine and treatments, and the MHC-PMS.

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

5.6. Look carefully at how messages and mailboxes are represented in the e-mail system that you use. Model the object classes that might be used in the system implementation to represent a mailbox and an e-mail message.

5.7. Based on your experience with a bank ATM, draw an activity diagram that models the data processing involved when a customer withdraws cash from the machine.

5.8. Draw a sequence diagram for the same system. Explain why you might want to develop both activity and sequence diagrams when modeling the behavior of a system.

5.9. Draw state diagrams of the control software for:

- An automatic washing machine that has different programs for different types of clothes.
- The software for a DVD player.
- A telephone answering system that records incoming messages and displays the number of accepted messages on an LED. The system should allow the telephone customer to dial in from any location, type a sequence of numbers (identified as tones), and play any recorded messages.

5.10. You are a software engineering manager and your team proposes that model-driven engineering should be used to develop a new system. What factors should you take into account when deciding whether or not to introduce this new approach to software development?

REFERENCES

Ambler, S. W. and Jeffries, R. (2002). *Agile Modeling: Effective Practices for Extreme Programming and the Unified Process*. New York: John Wiley & Sons.

Booch, G., Rumbaugh, J. and Jacobson, I. (2005). *The Unified Modeling Language User Guide, 2nd ed*. Boston: Addison-Wesley.

Chen, P. (1976). 'The entity relationship model—Towards a unified view of data'. *ACM Trans. on Database Systems*, **1** (1), 9–36.

Codd, E. F. (1979). 'Extending the database relational model to capture more meaning'. *ACM Trans. on Database Systems*, **4** (4), 397–434.

DeMarco, T. (1978). *Structured Analysis and System Specification*. New York: Yourdon Press.

Erickson, J. and Siau, K. (2007). 'Theoretical and practical complexity of modeling methods'. *Comm. ACM*, **50** (8), 46–51.

Hammer, M. and McLeod, D. (1981). 'Database descriptions with SDM: A semantic database model'. *ACM Trans. on Database Sys.*, **6** (3), 351–86.

Harel, D. (1987). 'Statecharts: A visual formalism for complex systems'. *Sci. Comput. Programming*, **8** (3), 231–74.

Harel, D. (1988). 'On visual formalisms'. *Comm. ACM*, **31** (5), 514–30.

Hull, R. and King, R. (1987). 'Semantic database modeling: Survey, applications and research issues'. *ACM Computing Surveys,* **19** (3), 201–60.

Jacobson, I., Christerson, M., Jonsson, P. and Overgaard, G. (1993). *Object-Oriented Software Engineering*. Wokingham.: Addison-Wesley.

Kent, S. (2002). 'Model-driven engineering'. Proc. 3rd Int. Conf. on Integrated Formal Methods, 286–98.

Kleppe, A., Warmer, J. and Bast, W. (2003). *MDA Explained: The Model Driven Architecture—Practice and Promise*. Boston: Addison-Wesley.

Kruchten, P. (1995). 'The 4 + 1 view model of architecture'. *IEEE Software*, **11** (6), 42–50.

Mellor, S. J. and Balcer, M. J. (2002). *Executable UML*. Boston: Addison-Wesley.

Mellor, S. J., Scott, K. and Weise, D. (2004). *MDA Distilled: Principles of Model-driven Architecture*. Boston: Addison-Wesley.

Rumbaugh, J., Jacobson, I. and Booch, G. (1999). *The Unified Modeling Language Reference Manual*. Reading, Mass.: Addison-Wesley.

Rumbaugh, J., Jacobson, I. and Booch, G. (2004). *The Unified Modeling Language Reference Manual, 2nd ed*. Boston: Addison-Wesley.

Schmidt, D. C. (2006). 'Model-Driven Engineering'. *IEEE Computer,* **39** (2), 25–31.

Stahl, T. and Voelter, M. (2006). *Model-Driven Software Development: Technology, Engineering, Management*. New York: John Wiley & Sons.

Ward, P. and Mellor, S. (1985). *Structured Development for Real-time Systems*. Englewood Cliffs, NJ: Prentice Hall.