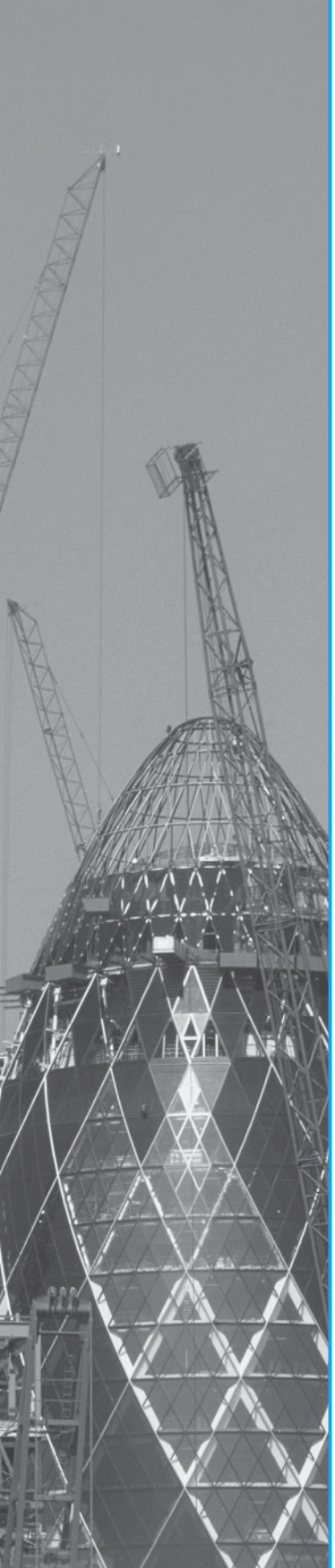

5 System modeling

Objectives

The aim of this chapter is to introduce system models that may be developed as part of 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 and why several types of model are needed to fully represent a system;
- understand the fundamental system modeling perspectives of context, interaction, structure, and behavior;
- understand the principal diagram types in the Unified Modeling Language (UML) and how these diagrams may be used in system modeling;
- have been introduced to model-driven engineering, where an executable 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 now usually means representing a system using some kind of graphical notation based on diagram types 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 here, and formal modeling is briefly discussed in Chapter 10.

Models are used during the requirements engineering process to help derive the detailed 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 they can be used to focus a stakeholder discussion on its strengths and weaknesses.
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. If you use a model-driven engineering process (Brambilla, Cabot, and Wimmer 2012), you can generate a complete or partial system implementation from system models.

It is important to understand that a system model is not a complete representation of system. It purposely leaves out detail to make it easier to understand. A model is an abstraction of the system being studied rather than an alternative representation of that system. A representation of a system should maintain all the information about the entity being represented. An abstraction deliberately simplifies a system design and picks out the most salient characteristics. For example, the PowerPoint slides that accompany this book are an abstraction of the book's key points. However, if the book 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 processed by the system.
4. A behavioral perspective, where you model the dynamic behavior of the system and how it responds to events.

The Unified Modeling Language

The Unified Modeling Language (UML) 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, such as SysML, have been proposed for more general system modeling.

http://software-engineering-book.com/web/uml/

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 depend on how you intend to use it. There are three ways in which graphical models are commonly used:

1. As a way to stimulate and focus discussion about an existing or proposed system. The purpose of the model is to stimulate and focus discussion among the software engineers involved in developing the system. The models may be incomplete (as 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 agile modeling (Ambler and Jeffries 2002).
2. As a way of documenting an existing system. When models are used as documentation, they do not have to be complete, as you may only need to use models to document 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.
3. As a detailed system description that can be used to generate a system implementation. Where models are used as part of a model-based development process, the system models have to be both complete and correct. They are used as a basis for generating the source code of the system, and you therefore have to be very careful not to confuse similar symbols, such as stick and block arrowheads, that may have different meanings.

In this chapter, I use diagrams defined in the Unified Modeling Language (UML) (Rumbaugh, Jacobson, and Booch 2004; Booch, Rumbaugh, and Jacobson 2005), which has become a standard language for object-oriented modeling. The UML has 13 diagram types and so supports the creation of many different types of system model. However, a survey (Erickson and Siau 2007) showed that most users of the UML thought that five diagram types could represent the essentials of a system. I therefore concentrate on these five UML diagram types here:

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.

5.1 Context models

At an early stage in the specification of a system, you should decide on the system boundaries, that is, on what is and is not part of the system being developed. This involves working with system stakeholders to decide what functionality should be included in the system and what processing and operations should be carried out in the system's operational environment. You may decide that automated support for some business processes should be implemented in the software being developed but that other processes should be manual 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 Mentcare patient information system. 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, and if these systems are unavailable, then it may be impossible to use the Mentcare system.

In some situations, the user base for a system is very diverse, and users have a wide range of different system requirements. You may decide not to define

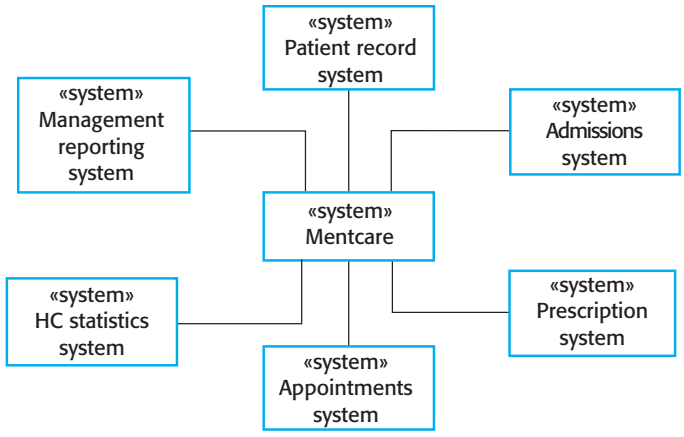

Figure 5.1 The context of the Mentcare system

boundaries explicitly but instead to develop a configurable system that can be adapted to the needs of different users. This was the approach that we adopted in the iLearn systems, introduced in Chapter 1. There, users range from very young children who can't read through to young adults, their teachers, and school administrators. Because these groups need different system boundaries, we specified a configuration system that would allow the boundaries to be specified when the system was deployed.

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 nontechnical factors. For example, a system boundary may be deliberately positioned so that the complete analysis process can be carried out on one site; it may be chosen so that a particularly difficult manager need not be consulted; and 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 context model that shows the Mentcare system and the other systems in its environment. You can see that the Mentcare system 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 admissions, 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

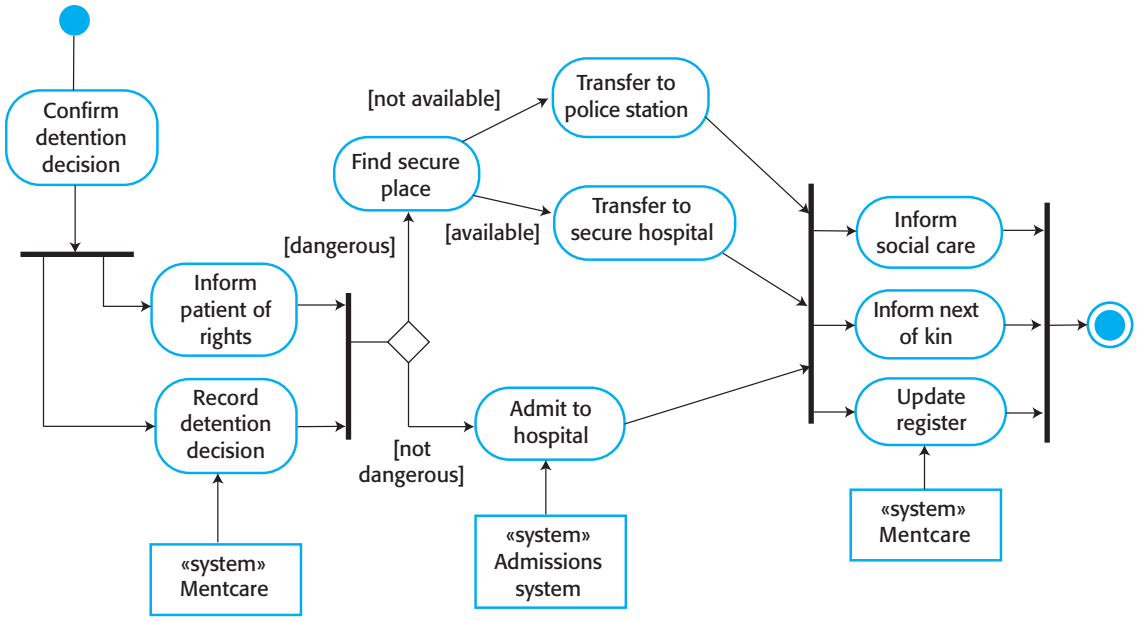

Figure 5.2 A process model of involuntary detention

these relations may affect the requirements and design of the system being defined and so 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.

UML activity diagrams may be used to show the business processes in which systems are used. Figure 5.2 is a UML activity diagram that shows where the Mentcare system is used in an important mental health care process—involuntary detention.

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 critical function of the Mentcare system is to ensure that such safeguards are implemented and that the rights of patients are respected.

UML activity diagrams show the activities in a 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 subprocesses that must be carried out. You may include objects in activity charts. Figure 5.2 shows the systems that are used to support different subprocesses within the involuntary detection process. I have shown that these are separate systems by using the UML stereotype feature where the type of entity in the box between chevrons is shown.

Arrows represent the flow of work from one activity to another, and a solid bar indicates activity coordination. When the flow from more than one activity leads to a

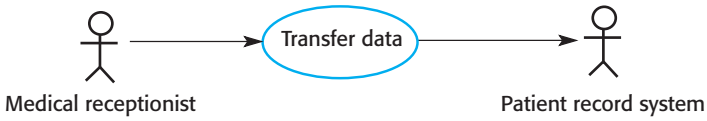

Figure 5.3 Transfer-data use case

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, as well as to update the detention register, may be concurrent.

Arrows may be annotated with guards (in square brackets) that specify when that flow is followed. 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 are a danger to themselves may be admitted to an appropriate ward in a hospital, where they can be kept under close supervision.

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 software being developed and other systems in its environment; or interaction between the components of a software system. User interaction modeling 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.

This section discusses two related approaches to interaction modeling:

1. Use case modeling, which is mostly used to model interactions between a system and external agents (human 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 interactions at different levels of detail and so may be used together. For example, 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 describe this diagram type because communication diagrams are simply an alternative representation of sequence diagrams.

5.2.1 Use case modeling

Use case modeling was originally developed by Ivar Jacobsen in the 1990s (Jacobsen et al. 1993), and a UML diagram type to support use case modeling is part of the

Mentcare system: Transfer data	
Actors	Medical receptionist, Patient records system (PRS)
Description	A receptionist may transfer data from the Mentcare system 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 5.4 Tabular description of the Transfer-data use case

UML. A use case can be taken as a simple description of what a user expects from a system in that interaction. I have discussed use cases for requirements elicitation in Chapter 4. As I said in Chapter 4, I find use case models to be more useful in the early stages of system design rather than in requirements engineering.

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 Mentcare system that represents the task of uploading data from the Mentcare system to a more general patient record system. This more general system maintains summary data about a patient rather than data about each consultation, which is recorded in the Mentcare system.

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 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 simple overview of an interaction, and you need to add more detail for complete interaction description. This detail can either be a simple textual description, a structured description in a table, or a sequence diagram. You choose 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.

Composite use case diagrams show a number of different use cases. Sometimes it is possible to include all possible interactions within 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 Mentcare system

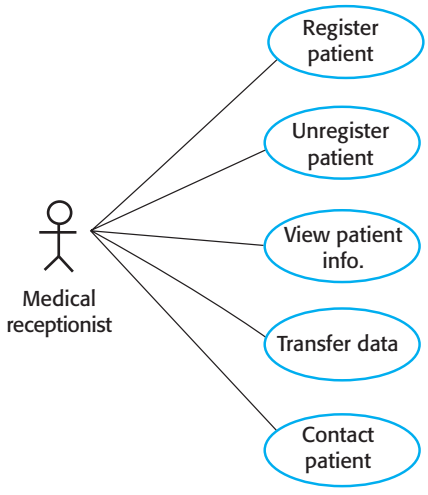

Figure 5.5 Use cases involving the role "Medical receptionist"

in which the actor "Medical Receptionist" is involved. Each of these should be accompanied by a more detailed description.

The UML includes a number of constructs for sharing all or part of a use case in other use case diagrams. While these constructs can sometimes be helpful for system designers, my experience is that many people, especially end-users, find them difficult to understand. For this reason, these constructs are not described here.

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. As space does not allow covering all possibilities here, the focus will be 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. Annotated arrows indicate interactions between objects. 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. This example also shows the notation used to denote alternatives. A box named alt is used with the

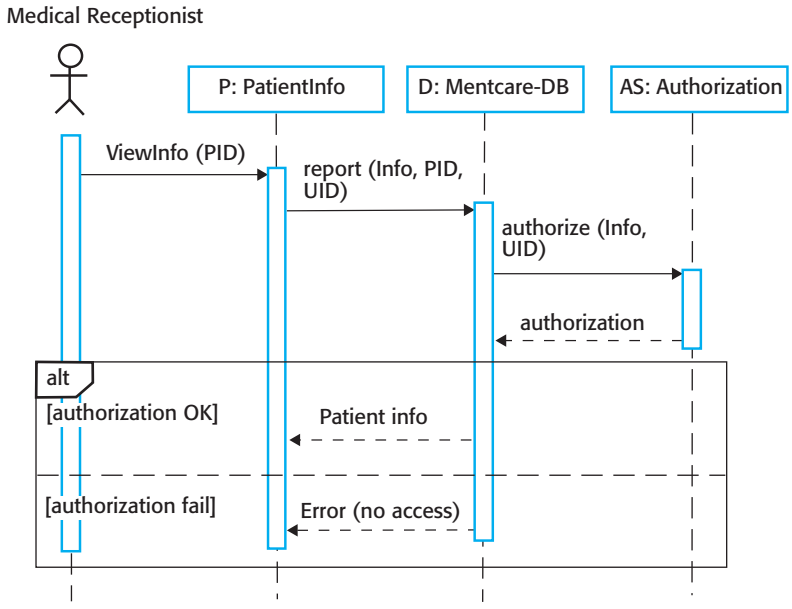

Figure 5.6 Sequence diagram for View patient information

conditions indicated in square brackets, with alternative interaction options separated by a dotted line.

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 to identify the required information. 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, it is not important where the receptionist's UID comes from.)
3. The database checks with an authorization system that the receptionist is authorized for this action.
4. If authorized, the patient information is returned and is displayed on a form on the user's screen. If authorization fails, then an error message is returned. The box denoted by "alt" in the top-left corner is a choice box indicating that one of the contained interactions will be executed. The condition that selects the choice is shown in square brackets.

Figure 5.7 is a further 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

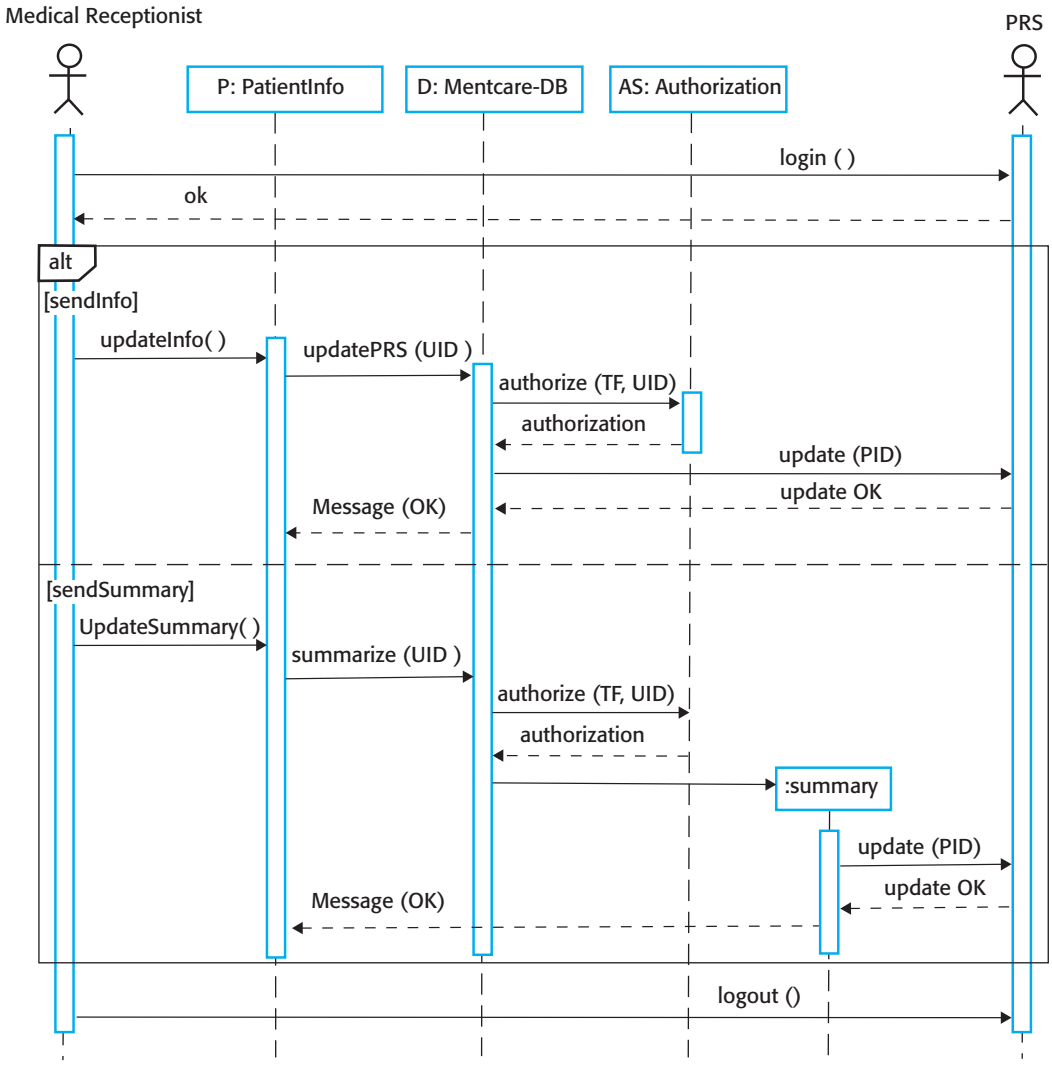

Figure 5.7 Sequence diagram for Transfer Data

to be uploaded to a national PRS (patient records system). You can read this diagram as follows:

1. The receptionist logs on to the PRS.
2. Two options are available (as shown in the "alt" box). These allow the direct transfer of updated patient information from the Mentcare database to the PRS and the transfer of summary health data from the Mentcare database 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.

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 that depend on implementation decisions. For example, in Figure 5.7 the decision on how to get the user identifier to check authorization is one that can be delayed. In an implementation, this might involve interacting with a User object. As this is not important at this stage, you do not need to include it in the sequence diagram.

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 organization 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. These can be models of the overall system architecture or more detailed models of the objects in the system and their relationships.

In this section, I focus on the use of class diagrams for modeling the static structure of the object classes in a software system. Architectural design is an important topic in software engineering, and UML component, package, and deployment diagrams may all be used when presenting architectural models. I cover architectural modeling in Chapters 6 and 17.

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 indicating that some relationship exists 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

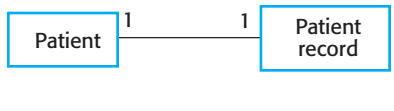

Figure 5.8 UML Classes and association

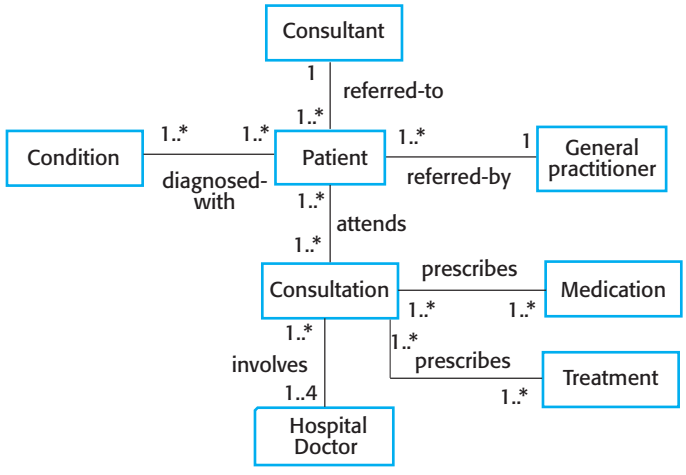

Figure 5.9 Classes and associations in the Mentcare system

prescription, or a doctor. As an implementation is developed, you define implementation objects to represent data that is manipulated by the system. In this section, the focus is on the modeling of real-world objects as part of the requirements or early software design processes. A similar approach is used for data structure modeling.

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 diagrams is to write the class name in a box. You can also note the existence of an 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. At this stage, you do not need to say what the association is.

Figure 5.9 develops the simple class diagram in Figure 5.8 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.

Figures 5.8 and 5.9, shows an important feature of class diagrams—the ability to show how many objects are involved in the association. In Figure 5.8 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 Figure 5.9, other multiplicities are possible. You can define that an exact number of objects are involved (e.g., 1..4) or, by using a *, indicate that there are an indefinite number of objects involved in the association. For example, the (1..*) multiplicity in Figure 5.9 on the relationship between Patient and Condition shows that a patient may suffer from several conditions and that the same condition may be associated with several patients.

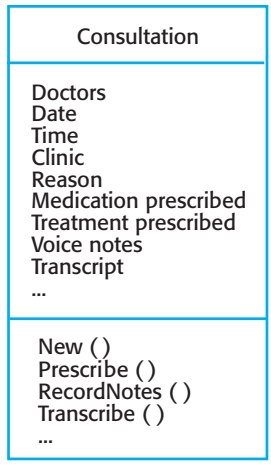

Figure 5.10 A Consultation class

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 (Hull and King 1987). The UML does not include a diagram type for database modeling, as it 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 (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 best to represent these classes in the simplest possible way, without attributes or operations. To define objects in more detail, you add information about their attributes (the object's characteristics) and operations (the object's functions). For example, a Patient object has the attribute Address, and you may include an operation called ChangeAddress, which is called when a patient indicates that he or she has moved from one address to another.

In the UML, you show attributes and operations by extending the simple rectangle that represents a class. I illustrate this in Figure 5.10 that shows an object representing a consultation between doctor and patient:

1. The name of the object class is in the top section.
2. The class attributes are in the middle section. This includes the attribute names and, optionally, their types. I don't show the types in Figure 5.10.
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. I show some but not all operations in Figure 5.10.

In the example shown in Figure 5.10, it is assumed 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.

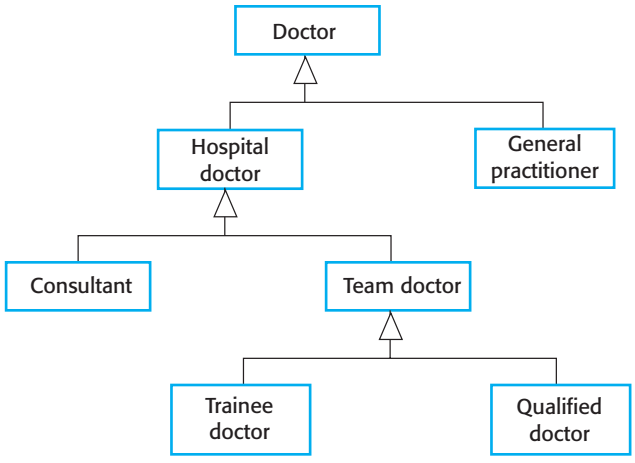

Figure 5.11 A generalization hierarchy

5.3.2 Generalization

Generalization is an everyday technique that we use to manage complexity. Rather than learn the detailed characteristics of everything that we experience, we learn about general classes (animals, cars, houses, etc.) and learn the characteristics of these classes. We then reuse knowledge by classifying things and focus on the differences between them and their class. For example, squirrels and rats are members of the class "rodents," and so share the characteristics of rodents. General statements apply to all class members; for example, all rodents have teeth for gnawing.

When you are modeling systems, it is often useful to examine the classes in a system to see if there is scope for generalization and class creation. 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. You can make the changes at the most general level. 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 indicates that general practitioners and hospital doctors can be generalized as doctors and that there are three types of Hospital Doctor: those who have just graduated from medical school and have to be supervised (Trainee Doctor); those who 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. The lower-level classes are subclasses that inherit the attributes and operations from their superclasses. These lower-level classes then add more specific attributes and operations.

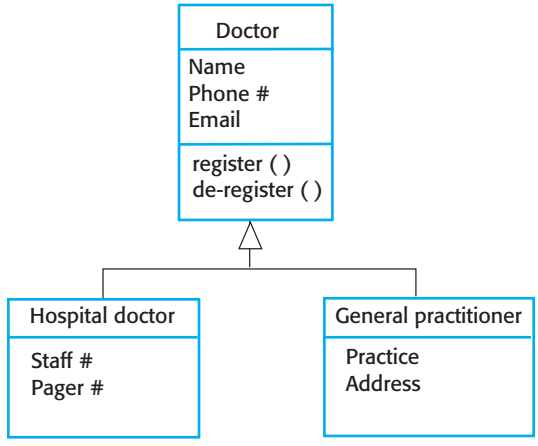

Figure 5.12 A generalization hierarchy with added detail

For example, all doctors have a name and phone number, and all hospital doctors have a staff number and carry a pager. General practitioners don't have these attributes, as they work independently, but they have an individual practice name and address. Figure 5.12 shows part of the generalization hierarchy, which I have extended with class attributes, for the class Doctor. The operations associated with the class Doctor are intended to register and de-register that doctor with the Mentcare system.

5.3.3 Aggregation

Objects in the real world are often made up 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, which means that one object (the whole) is composed of other objects (the parts). To define aggregation, a diamond shape is added to the link next to the class that represents the whole.

Figure 5.13 shows that a patient record is an aggregate of Patient and an indefinite number of Consultations. That is, the record maintains personal patient information as well as an individual record for each consultation with a doctor.

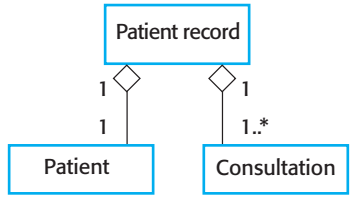

Figure 5.13 The aggregation association

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. Activity diagrams in the UML may be used to represent DFDs.

http://software-engineering-book.com/web/dfds/

5.4 Behavioral models

Behavioral models are models of the dynamic behavior of a 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. These stimuli may be either data or events:

1. Data becomes available that has to be processed by the system. The availability of the data triggers the processing.
2. An event happens that triggers system processing. Events may have associated data, although this is not always the case.

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 for that customer.

By contrast, real-time systems are usually event-driven, with limited data processing. For example, a landline phone switching system responds to events such as "handset activated" by generating a dial tone, pressing keys on a handset by capturing the phone number, and so on.

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 can be used during the analysis of requirements as they show end-to-end processing in a system. That is, they show the entire sequence of actions that takes place from an initial input being processed to the corresponding output, which is the system's response.

Data-driven models were among the first graphical software models. In the 1970s, structured design methods used data-flow diagrams (DFDs) as a way to illustrate the

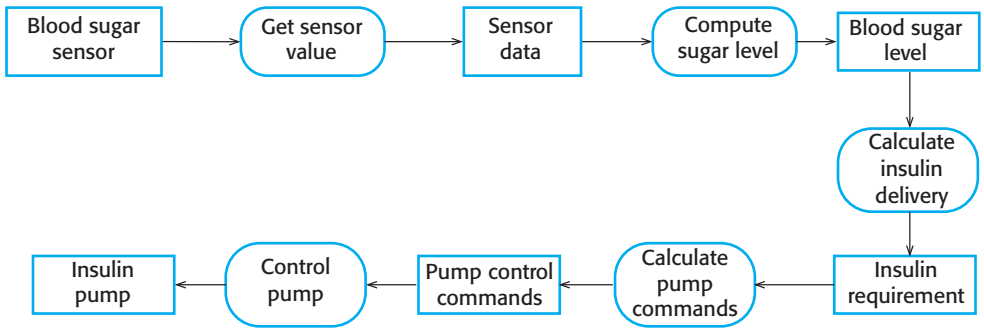

Figure 5.14 An activity model of the insulin pump's operation

processing steps in a system. Data-flow models are useful because tracking and documenting how data associated with a particular process moves through the system help analysts and designers understand what is going on in the process. DFDs are simple and intuitive and so are more accessible to stakeholders than some other types of model. It is usually possible to explain them to potential system users who can then participate in validating the model.

Data-flow diagrams can be represented in the UML using the activity diagram type, described in Section 5.1. Figure 5.14 is a simple activity diagram that shows the chain of processing involved in the insulin pump software. You can see the processing steps, represented as activities (rounded rectangles), and the data flowing between these steps, represented as objects (rectangles).

An alternative way of showing the sequence of processing in a system is to use UML sequence diagrams. You have seen how these diagrams can be used to model interaction, but if 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 processing an order and sending it to a supplier. Sequence models highlight objects in a system, whereas data-flow diagrams highlight the operations or activities. In practice, nonexperts seem to find data-flow diagrams more intuitive, but engineers prefer sequence diagrams.

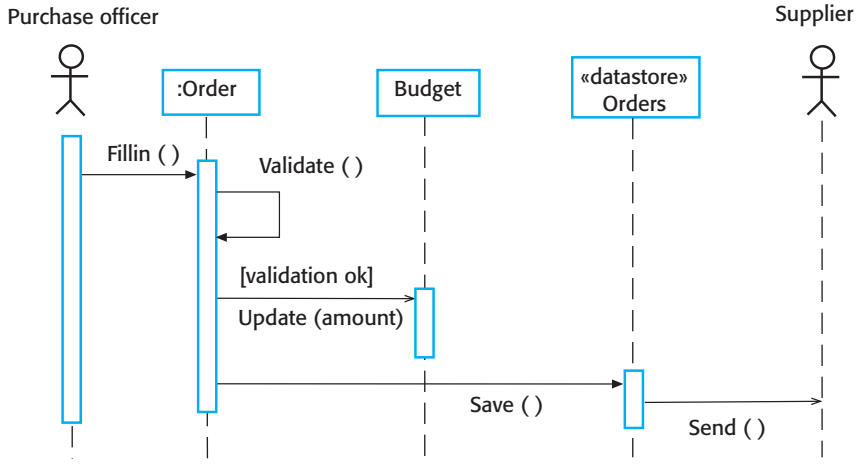

Figure 5.15 Order processing

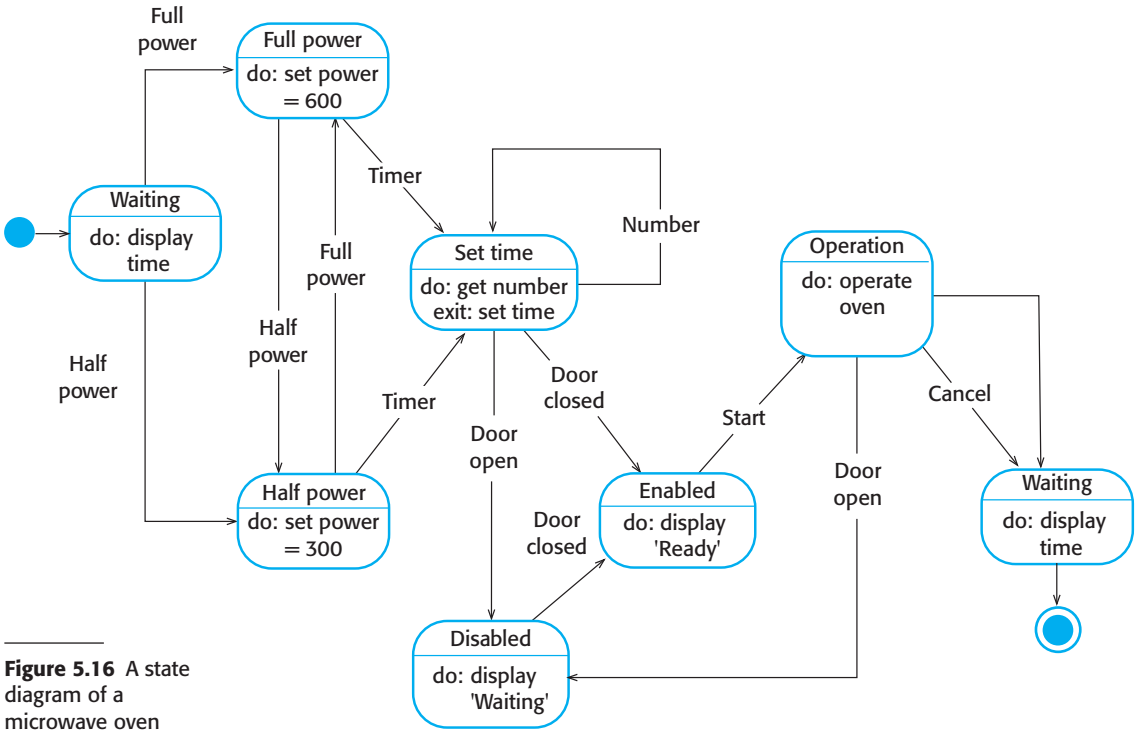

Figure 5.16 A state diagram of a microwave oven

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-driven modeling is used extensively when designing and documenting real-time systems (Chapter 21).

The UML supports event-based modeling using state diagrams, which are based on Statecharts (Harel 1987). State diagrams show system states and events that cause 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 (Figure 5.16). Real microwave ovens are much more complex than this system, but the simplified system is easier to understand. This simple oven 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.

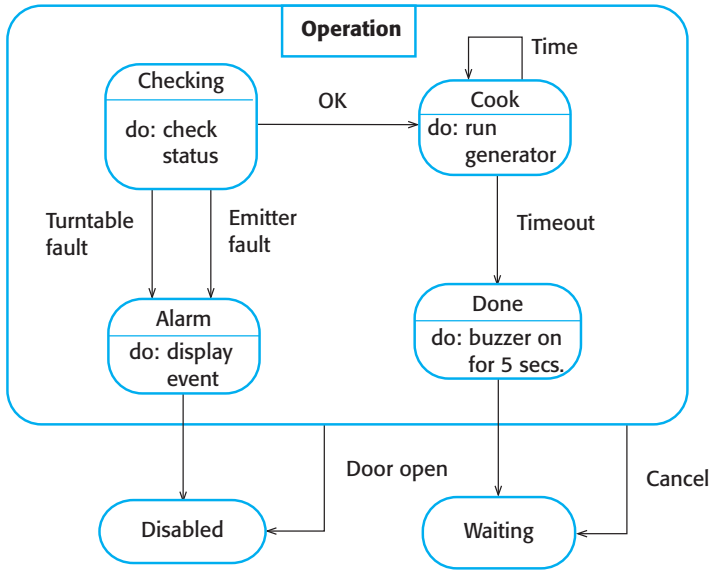

Figure 5.17 A state model of the Operation state

I have assumed that the sequence of actions in using the microwave is as follows:

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 simple 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 their minds after selecting one of these and may 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 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 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 5.16. This is a superstate that can be expanded, as shown in Figure 5.17.

State	Description
Waiting	The oven is waiting for input. The display shows the current time.
Half power	The oven power is set to 300 watts. The display shows "Half power."
Full power	The oven power is set to 600 watts. The display shows "Full power."
Set time	The cooking time is set to the user's input value. The display shows the cooking time selected and is updated as the time is set.
Disabled	Oven operation is disabled for safety. Interior oven light is on. Display shows "Not ready."
Enabled	Oven operation is enabled. Interior oven light is off. Display shows "Ready to cook."
Operation	Oven in operation. Interior oven light is on. Display shows the timer countdown. On completion of cooking, the buzzer is sounded for 5 seconds. Oven light is on. Display shows "Cooking complete" while buzzer is sounding.
Stimulus	**Description**
Half power	The user has pressed the half-power button.
Full power	The user has pressed the full-power button.
Timer	The user has pressed one of the timer buttons.
Number	The user has pressed a numeric key.
Door open	The oven door switch is not closed.
Door closed	The oven door switch is closed.
Start	The user has pressed the Start button.
Cancel	The user has pressed the Cancel button.

Figure 5.18 States and stimuli for the microwave oven

The Operation state includes a number of substates. 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 5.17.

State models of a system provide an overview of event processing, but you normally have to extend this with a more detailed description of the stimuli and the system states. You may use a table to list the states and events that stimulate state transitions along with a description of each state and event. Figure 5.18 shows a tabular description of each state and how the stimuli that force state transitions are generated.

5.4.3 Model-driven engineering

Model-driven engineering (MDE) is an approach to software development whereby models rather than programs are the principal outputs of the development process

(Brambilla, Cabot, and Wimmer 2012). The programs that execute on a hardware/software platform are 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 was developed from the idea of model-driven architecture (MDA). This was proposed by the Object Management Group (OMG) as a new software development paradigm (Mellor, Scott, and Weise 2004). 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 are not considered in MDA.

MDA as an approach to system engineering has been adopted by a number of large companies to support their development processes. This section focuses on the use of MDA for software implementation rather than discuss more general aspects of MDE. The take-up of more general model-driven engineering has been slow, and few companies have adopted this approach throughout their software development life cycle. In his blog, den Haan discusses possible reasons why MDE has not been widely adopted (den Haan 2011).

5.5 Model-driven architecture

Model-driven architecture (Mellor, Scott, and Weise 2004; Stahl and Voelter 2006) is a model-focused approach to software design and implementation that uses a subset of UML models to describe a system. Here, models at different 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)* CIMs model the important domain abstractions used in a system and so 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 and consultations.

2. *A platform-independent model (PIM)* PIMs model the operation of the system without reference to its implementation. A PIM is usually described using UML models that show the static system structure and how it responds to external and internal events.

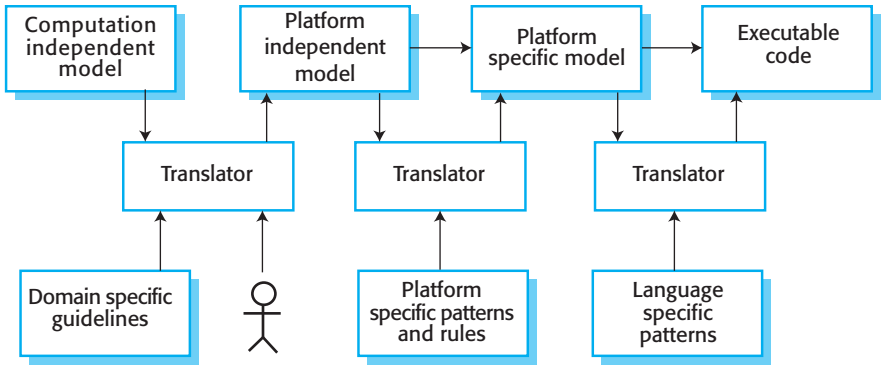

Figure 5.19 MDA transformations

3. *Platform-specific models (PSM)* PSMs 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.

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, you write a model translator for that platform. When this is available, all platform-independent models can then be rapidly re-hosted on the new platform.

Fundamental to MDA is the notion that transformations between models can be defined and applied automatically by software tools, as illustrated in Figure 5.19. This diagram also shows a final level of automatic transformation where a transformation is applied to the PSM to generate the executable code that will run on the designated software platform. Therefore, in principle at least, executable software can be generated from a high-level system model.

In practice, completely automated translation of models to code is rarely possible. The translation of high-level CIM to PIM models remains a research problem, and for production systems, human intervention, illustrated using a stick figure in Figure 5.19, is normally required. A particularly difficult problem for automated model transformation is the need to link the concepts used in different CIMS. For example, the concept of a role in a security CIM that includes role-driven access control may have to be mapped onto the concept of a staff member in a hospital CIM. Only a person who understands both security and the hospital environment can make this mapping.

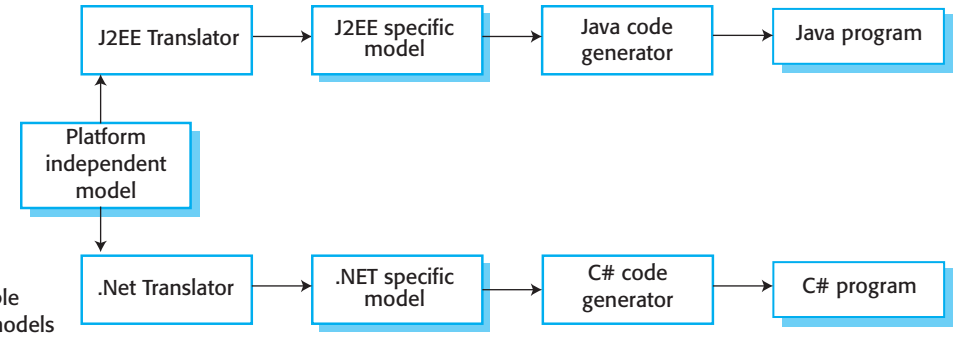

Figure 5.20 Multiple platform-specific models

The translation of platform-independent to platform-specific models is a simpler technical problem. Commercial tools and open-source tools (Koegel 2012) are available that provide translators from PIMS to common platforms such as Java and J2EE. These use an extensive library of platform-specific rules and patterns to convert a PIM to a 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, in principle, you only have to maintain a single PIM. The PSMs for each platform are automatically generated (Figure 5.20).

Although MDA support tools include platform-specific translators, these sometimes only offer partial support for translating PIMS to PSMs. The execution environment for a system is more than the standard execution platform, such as J2EE or Java. It also includes other application systems, specific application libraries that may be created for a company, external services, and user interface libraries.

These vary from one company to another, so off-the-shelf tool support is not available that takes these into account. Therefore, when MDA is introduced into an organization, special-purpose translators may have to be created to make use of the facilities available in the local environment. This is one reason why many companies have been reluctant to take on model-driven approaches to development. They do not want to develop or maintain their own tools or to rely on small software companies, who may go out of business, for tool development. Without these specialist tools, model-based development requires additional manual coding which reduces the cost-effectiveness of this approach.

I believe that there are several other reasons why MDA has not become a mainstream approach to software development.

1. Models are a good way of facilitating discussions about a software design. However, it does not always follow that the abstractions that are useful for discussions are the right abstractions for implementation. You may decide to use a completely different implementation approach that is based on the reuse of off-the-shelf application systems.

2. For most complex systems, implementation is not the major problem—requirements engineering, security and dependability, integration with legacy

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 with clearly defined meanings that can be compiled to executable code. 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).

http://software-engineering-book.com/web/xuml/

systems and testing are all more significant. Consequently, the gains from the use of MDA are limited.

3. The arguments for platform independence are only valid for large, long-lifetime systems, where the platforms become obsolete during a system's lifetime. For software products and information systems that are developed for standard platforms, such as Windows and Linux, the savings from the use of MDA are likely to be outweighed by the costs of its introduction and tooling.
4. The widespread adoption of agile methods over the same period that MDA was evolving has diverted attention away from model-driven approaches.

The success stories for MDA (OMG 2012) have mostly come from companies that are developing systems products, which include both hardware and software. The software in these products has a long lifetime and may have to be modified to reflect changing hardware technologies. The domain of application (automotive, air traffic control, etc.) is often well understood and so can be formalized in a CIM.

Hutchinson and his colleagues (Hutchinson, Rouncefield, and Whittle 2012) report on the industrial use of MDA, and their work confirms that successes in the use of model-driven development have been in systems products. Their assessment suggests that companies have had mixed results when adopting this approach, but the majority of users report that using MDA has increased productivity and reduced maintenance costs. They found that MDA was particularly useful in facilitating reuse, and this led to major productivity improvements.

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. Ambler, a pioneer in the development of agile methods, suggests that some aspects of MDA can be used in agile processes (Ambler 2004) but considers automated code generation to be impractical. However, Zhang and Patel report on Motorola's success in using agile development with automated code generation (Zhang and Patel 2011).