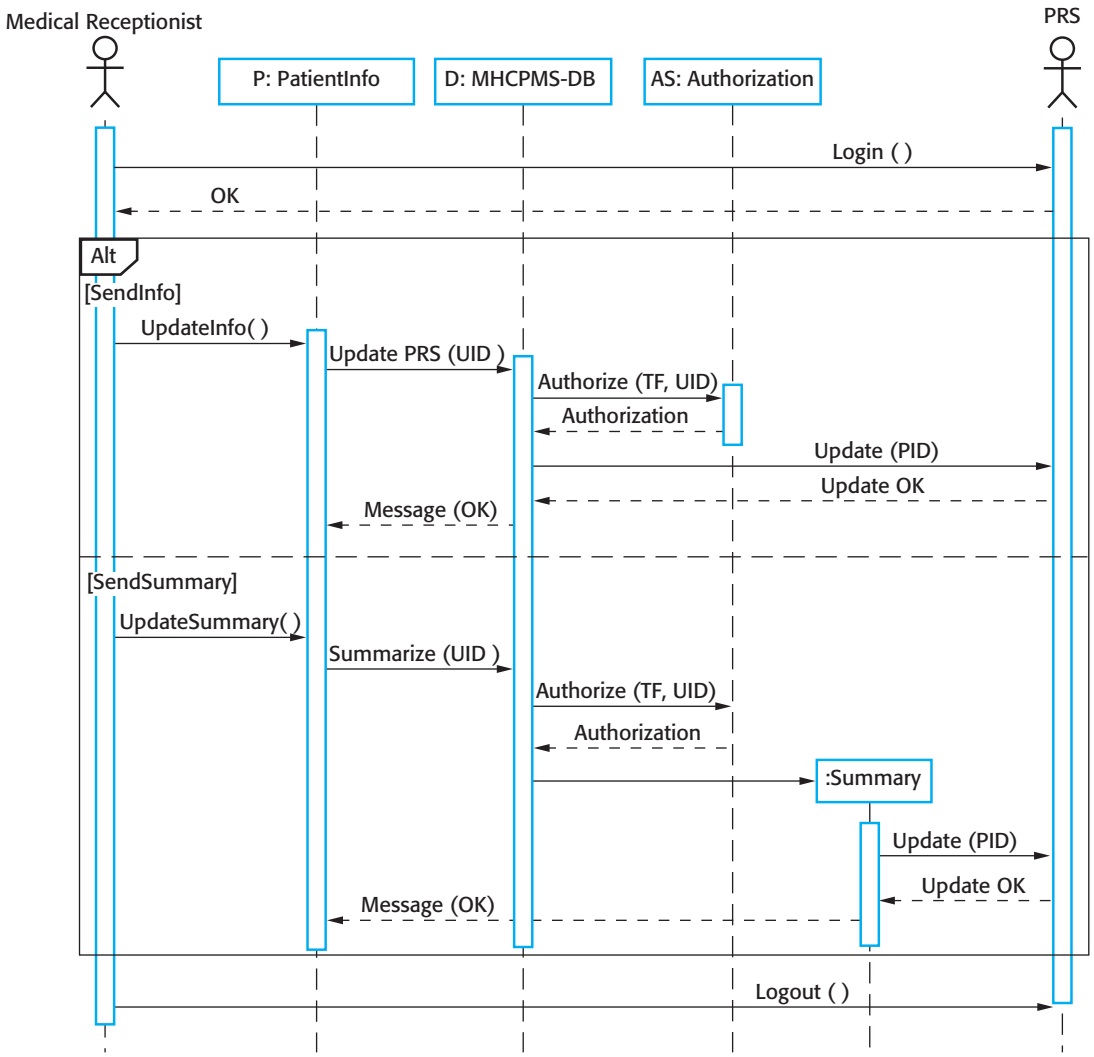


Figure 5.7 Sequence diagram for transfer data

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

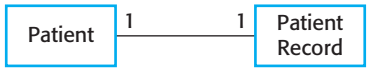


Figure 5.8 UML classes and association

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

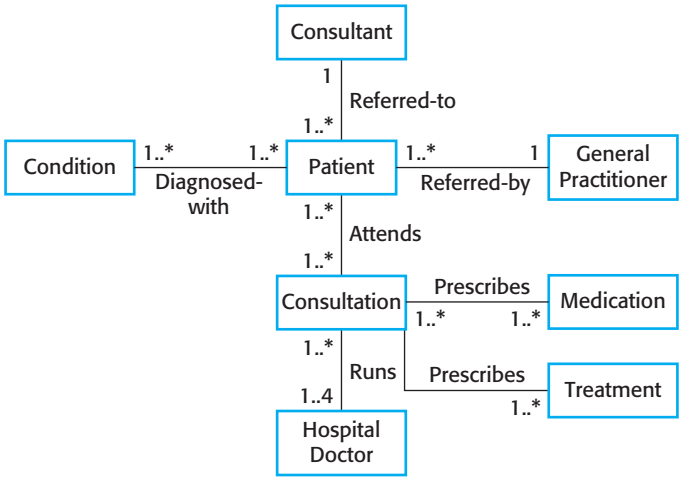


Figure 5.9 Classes and associations in the MHC-PMS

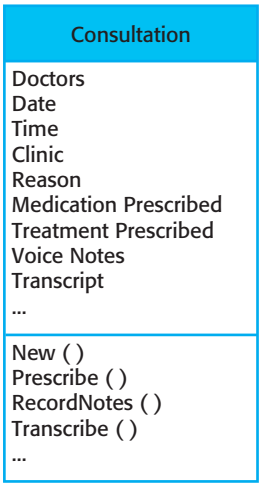


Figure 5.10 The consultation class

(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

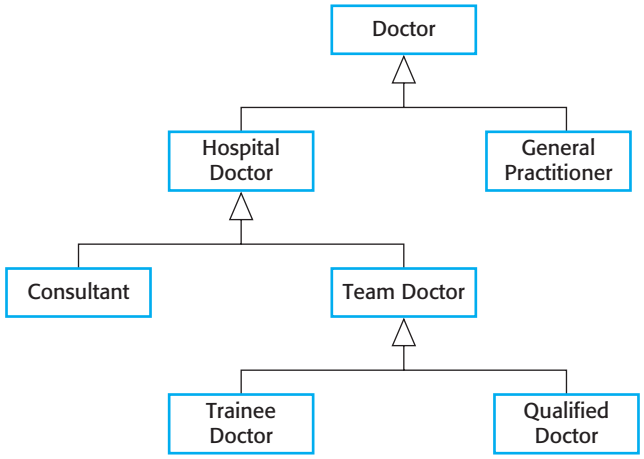


Figure 5.11 A generalization hierarchy

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.

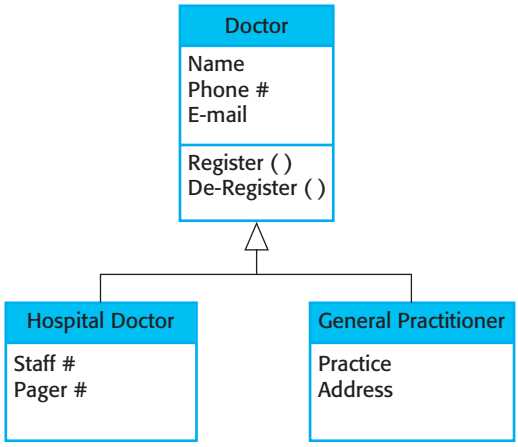


Figure 5.12 A generalization hierarchy with added detail

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.

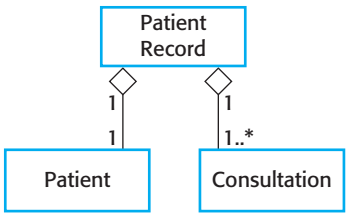


Figure 5.13 The aggregation association

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

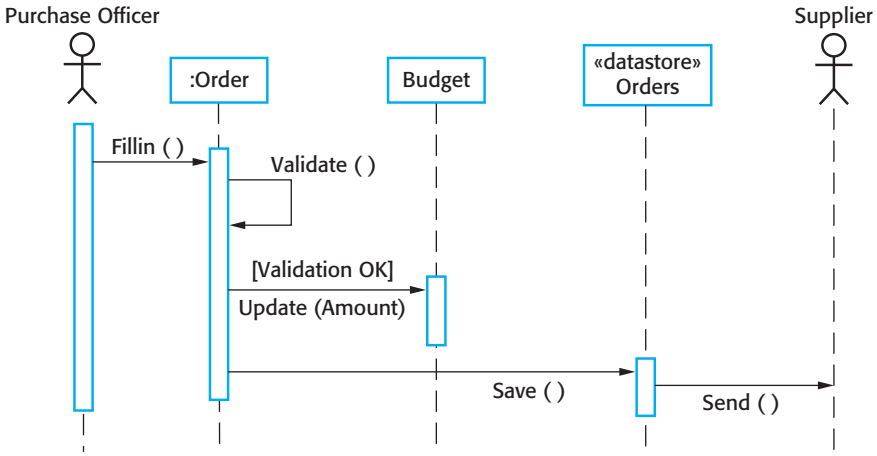


Figure 5.14 An activity model of the insulin pump's operation

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

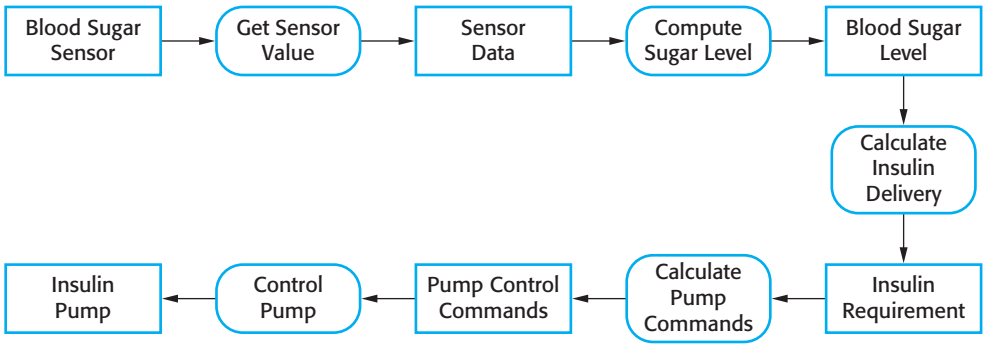


Figure 5.15 Order processing

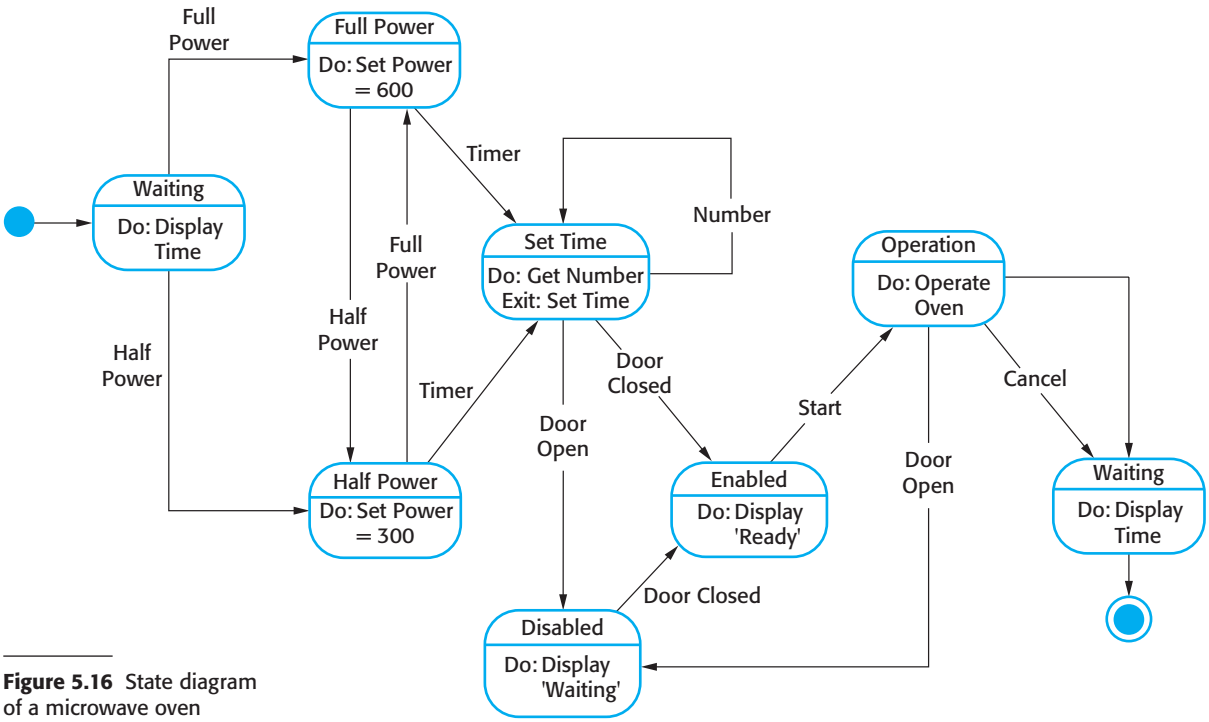


Figure 5.16 State diagram of a microwave oven

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

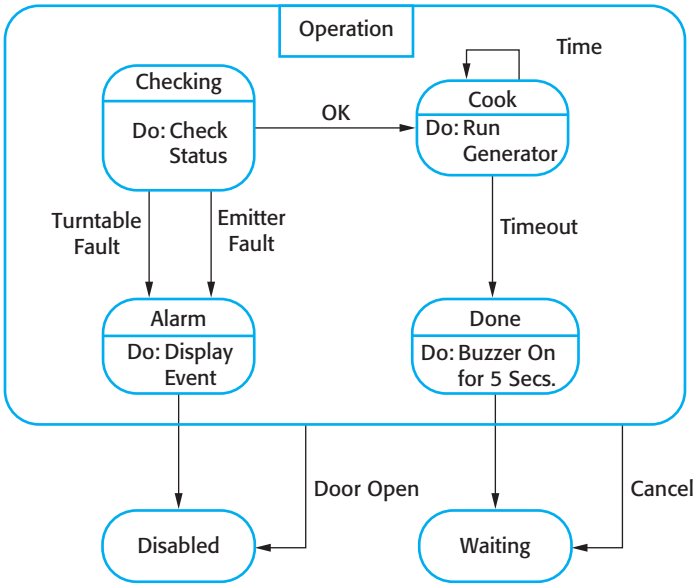


Figure 5.18 Microwave oven operation

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

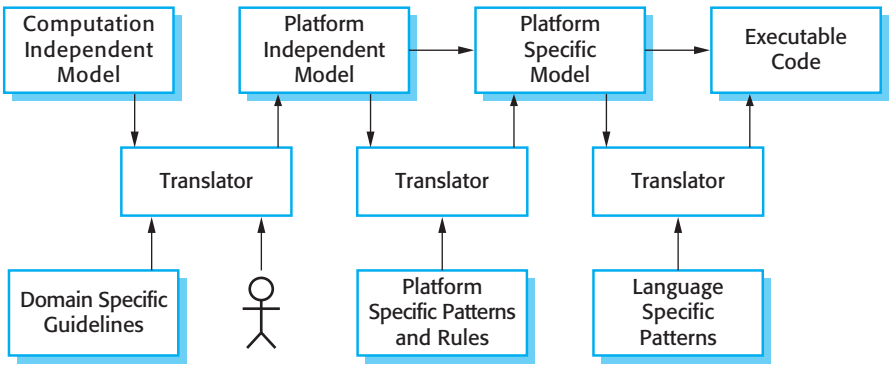


Figure 5.19 MDA transformations

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

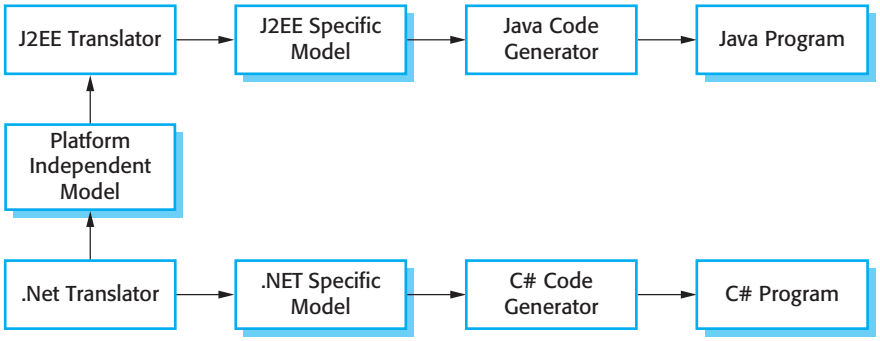


Figure 5.20 Multiple platform-specific models

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.

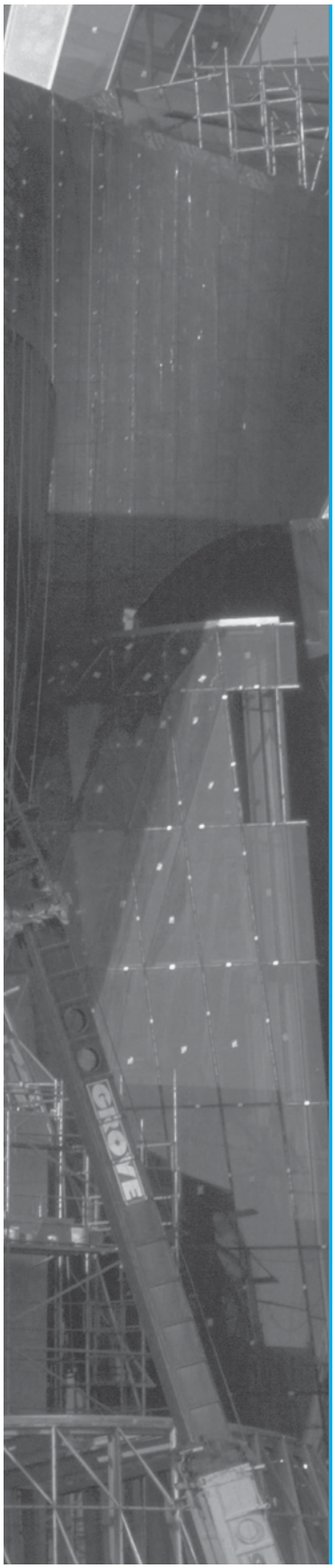
Rumbaugh, J., Jacobson, I. and Booch, G. (1999). *The Unified Modeling Language Reference Manual*. Reading, Mass.: Addison-Wesley.

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.



6 Architectural design

Objectives

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

- understand why the architectural design of software is important;
- understand the decisions that have to be made about the system architecture during the architectural design process;
- have been introduced to the idea of architectural patterns, well-tried ways of organizing system architectures, which can be reused in system designs;
- know the architectural patterns that are often used in different types of application system, including transaction processing systems and language processing systems.

Contents

Architectural design is concerned with understanding how a system should be organized and designing the overall structure of that system. In the model of the software development process, as shown in Chapter 2, architectural design is the first stage in the software design process. It is the critical link between design and requirements engineering, as it identifies the main structural components in a system and the relationships between them. The output of the architectural design process is an architectural model that describes how the system is organized as a set of communicating components.

In agile processes, it is generally accepted that an early stage of the development process should be concerned with establishing an overall system architecture. Incremental development of architectures is not usually successful. While refactoring components in response to changes is usually relatively easy, refactoring a system architecture is likely to be expensive.

To help you understand what I mean by system architecture, consider Figure 6.1. This shows an abstract model of the architecture for a packing robot system that shows the components that have to be developed. This robotic system can pack different kinds of object. It uses a vision component to pick out objects on a conveyor, identify the type of object, and select the right kind of packaging. The system then moves objects from the delivery conveyor to be packaged. It places packaged objects on another conveyor. The architectural model shows these components and the links between them.

In practice, there is a significant overlap between the processes of requirements engineering and architectural design. Ideally, a system specification should not include any design information. This is unrealistic except for very small systems. Architectural decomposition is usually necessary to structure and organize the specification. Therefore, as part of the requirements engineering process, you might propose an abstract system architecture where you associate groups of system functions or features with large-scale components or sub-systems. You can then use this decomposition to discuss the requirements and features of the system with stakeholders.

You can design software architectures at two levels of abstraction, which I call *architecture in the small* and *architecture in the large*:

1. Architecture in the small is concerned with the architecture of individual programs. At this level, we are concerned with the way that an individual program is decomposed into components. This chapter is mostly concerned with program architectures.
2. Architecture in the large is concerned with the architecture of complex enterprise systems that include other systems, programs, and program components. These enterprise systems are distributed over different computers, which may be owned and managed by different companies. I cover architecture in the large in Chapters 18 and 19, where I discuss distributed systems architectures.

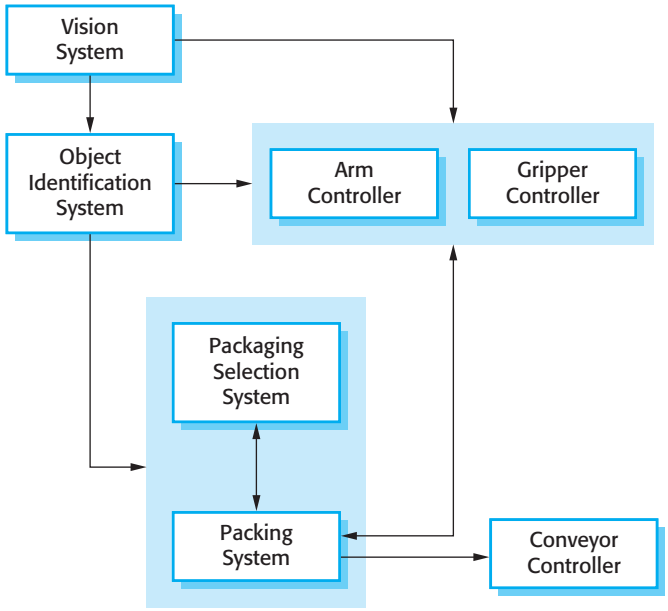


Figure 6.1 The architecture of a packing robot control system

Software architecture is important because it affects the performance, robustness, distributability, and maintainability of a system (Bosch, 2000). As Bosch discusses, individual components implement the functional system requirements. The non-functional requirements depend on the system architecture—the way in which these components are organized and communicate. In many systems, non-functional requirements are also influenced by individual components, but there is no doubt that the architecture of the system is the dominant influence.

Bass et al. (2003) discuss three advantages of explicitly designing and documenting software architecture:

1. *Stakeholder communication* The architecture is a high-level presentation of the system that may be used as a focus for discussion by a range of different stakeholders.
2. *System analysis* Making the system architecture explicit at an early stage in the system development requires some analysis. Architectural design decisions have a profound effect on whether or not the system can meet critical requirements such as performance, reliability, and maintainability.
3. *Large-scale reuse* A model of a system architecture is a compact, manageable description of how a system is organized and how the components interoperate. The system architecture is often the same for systems with similar requirements and so can support large-scale software reuse. As I explain in Chapter 16, it may be possible to develop product-line architectures where the same architecture is reused across a range of related systems.

Hofmeister et al. (2000) propose that a software architecture can serve firstly as a design plan for the negotiation of system requirements, and secondly as a means of structuring discussions with clients, developers, and managers. They also suggest that it is an essential tool for complexity management. It hides details and allows the designers to focus on the key system abstractions.

System architectures are often modeled using simple block diagrams, as in Figure 6.1. Each box in the diagram represents a component. Boxes within boxes indicate that the component has been decomposed to sub-components. Arrows mean that data and or control signals are passed from component to component in the direction of the arrows. You can see many examples of this type of architectural model in Booch's software architecture catalog (Booch, 2009).

Block diagrams present a high-level picture of the system structure, which people from different disciplines, who are involved in the system development process, can readily understand. However, in spite of their widespread use, Bass et al. (2003) dislike informal block diagrams for describing an architecture. They claim that these informal diagrams are poor architectural representations, as they show neither the type of the relationships among system components nor the components' externally visible properties.

The apparent contradictions between practice and architectural theory arise because there are two ways in which an architectural model of a program is used:

1. *As a way of facilitating discussion about the system design* A high-level architectural view of a system is useful for communication with system stakeholders and project planning because it is not cluttered with detail. Stakeholders can relate to it and understand an abstract view of the system. They can then discuss the system as a whole without being confused by detail. The architectural model identifies the key components that are to be developed so managers can start assigning people to plan the development of these systems.
2. *As a way of documenting an architecture that has been designed* The aim here is to produce a complete system model that shows the different components in a system, their interfaces, and their connections. The argument for this is that such a detailed architectural description makes it easier to understand and evolve the system.

Block diagrams are an appropriate way of describing the system architecture during the design process, as they are a good way of supporting communications between the people involved in the process. In many projects, these are often the only architectural documentation that exists. However, if the architecture of a system is to be thoroughly documented then it is better to use a notation with well-defined semantics for architectural description. However, as I discuss in Section 6.2, some people think that detailed documentation is neither useful, nor really worth the cost of its development.

6.1 Architectural design decisions

Architectural design is a creative process where you design a system organization that will satisfy the functional and non-functional requirements of a system. Because it is a creative process, the activities within the process depend on the type of system being developed, the background and experience of the system architect, and the specific requirements for the system. It is therefore useful to think of architectural design as a series of decisions to be made rather than a sequence of activities.

During the architectural design process, system architects have to make a number of structural decisions that profoundly affect the system and its development process. Based on their knowledge and experience, they have to consider the following fundamental questions about the system:

1. Is there a generic application architecture that can act as a template for the system that is being designed?
2. How will the system be distributed across a number of cores or processors?
3. What architectural patterns or styles might be used?
4. What will be the fundamental approach used to structure the system?
5. How will the structural components in the system be decomposed into subcomponents?
6. What strategy will be used to control the operation of the components in the system?
7. What architectural organization is best for delivering the non-functional requirements of the system?
8. How will the architectural design be evaluated?
9. How should the architecture of the system be documented?

Although each software system is unique, systems in the same application domain often have similar architectures that reflect the fundamental concepts of the domain. For example, application product lines are applications that are built around a core architecture with variants that satisfy specific customer requirements. When designing a system architecture, you have to decide what your system and broader application classes have in common, and decide how much knowledge from these application architectures you can reuse. I discuss generic application architectures in Section 6.4 and application product lines in Chapter 16.

For embedded systems and systems designed for personal computers, there is usually only a single processor and you will not have to design a distributed architecture for the system. However, most large systems are now distributed systems in which the system software is distributed across many different computers. The choice of distribution architecture is a key decision that affects the performance and

reliability of the system. This is a major topic in its own right and I cover it separately in Chapter 18.

The architecture of a software system may be based on a particular architectural pattern or style. An architectural pattern is a description of a system organization (Garlan and Shaw, 1993), such as a client–server organization or a layered architecture. Architectural patterns capture the essence of an architecture that has been used in different software systems. You should be aware of common patterns, where they can be used, and their strengths and weaknesses when making decisions about the architecture of a system. I discuss a number of frequently used patterns in Section 6.3.

Garlan and Shaw's notion of an architectural style (style and pattern have come to mean the same thing) covers questions 4 to 6 in the previous list. You have to choose the most appropriate structure, such as client–server or layered structuring, that will enable you to meet the system requirements. To decompose structural system units, you decide on the strategy for decomposing components into sub-components. The approaches that you can use allow different types of architecture to be implemented. Finally, in the control modeling process, you make decisions about how the execution of components is controlled. You develop a general model of the control relationships between the various parts of the system.

Because of the close relationship between non-functional requirements and software architecture, the particular architectural style and structure that you choose for a system should depend on the non-functional system requirements:

1. *Performance* If performance is a critical requirement, the architecture should be designed to localize critical operations within a small number of components, with these components all deployed on the same computer rather than distributed across the network. This may mean using a few relatively large components rather than small, fine-grain components, which reduces the number of component communications. You may also consider run-time system organizations that allow the system to be replicated and executed on different processors.
2. *Security* If security is a critical requirement, a layered structure for the architecture should be used, with the most critical assets protected in the innermost layers, with a high level of security validation applied to these layers.
3. *Safety* If safety is a critical requirement, the architecture should be designed so that safety-related operations are all located in either a single component or in a small number of components. This reduces the costs and problems of safety validation and makes it possible to provide related protection systems that can safely shut down the system in the event of failure.
4. *Availability* If availability is a critical requirement, the architecture should be designed to include redundant components so that it is possible to replace and update components without stopping the system. I describe two fault-tolerant system architectures for high-availability systems in Chapter 13.
5. *Maintainability* If maintainability is a critical requirement, the system architecture should be designed using fine-grain, self-contained components that may

readily be changed. Producers of data should be separated from consumers and shared data structures should be avoided.

Obviously there is potential conflict between some of these architectures. For example, using large components improves performance and using small, fine-grain components improves maintainability. If both performance and maintainability are important system requirements, then some compromise must be found. This can sometimes be achieved by using different architectural patterns or styles for different parts of the system.

Evaluating an architectural design is difficult because the true test of an architecture is how well the system meets its functional and non-functional requirements when it is in use. However, you can do some evaluation by comparing your design against reference architectures or generic architectural patterns. Bosch's (2000) description of the non-functional characteristics of architectural patterns can also be used to help with architectural evaluation.

6.2 Architectural views

I explained in the introduction to this chapter that architectural models of a software system can be used to focus discussion about the software requirements or design. Alternatively, they may be used to document a design so that it can be used as a basis for more detailed design and implementation, and for the future evolution of the system. In this section, I discuss two issues that are relevant to both of these:

1. What views or perspectives are useful when designing and documenting a system's architecture?
2. What notations should be used for describing architectural models?

It is impossible to represent all relevant information about a system's architecture in a single architectural model, as each model only shows one view or perspective of the system. It might show how a system is decomposed into modules, how the run-time processes interact, or the different ways in which system components are distributed across a network. All of these are useful at different times so, for both design and documentation, you usually need to present multiple views of the software architecture.

There are different opinions as to what views are required. Krutchen (1995), in his well-known 4+1 view model of software architecture, suggests that there should be four fundamental architectural views, which are related using use cases or scenarios. The views that he suggests are:

1. A logical view, which shows the key abstractions in the system as objects or object classes. It should be possible to relate the system requirements to entities in this logical view.

2. A process view, which shows how, at run-time, the system is composed of interacting processes. This view is useful for making judgments about non-functional system characteristics such as performance and availability.
3. A development view, which shows how the software is decomposed for development, that is, it shows the breakdown of the software into components that are implemented by a single developer or development team. This view is useful for software managers and programmers.
4. A physical view, which shows the system hardware and how software components are distributed across the processors in the system. This view is useful for systems engineers planning a system deployment.

Hofmeister et al. (2000) suggest the use of similar views but add to this the notion of a conceptual view. This view is an abstract view of the system that can be the basis for decomposing high-level requirements into more detailed specifications, help engineers make decisions about components that can be reused, and represent a product line (discussed in Chapter 16) rather than a single system. Figure 6.1, which describes the architecture of a packing robot, is an example of a conceptual system view.

In practice, conceptual views are almost always developed during the design process and are used to support architectural decision making. They are a way of communicating the essence of a system to different stakeholders. During the design process, some of the other views may also be developed when different aspects of the system are discussed, but there is no need for a complete description from all perspectives. It may also be possible to associate architectural patterns, discussed in the next section, with the different views of a system.

There are differing views about whether or not software architects should use the UML for architectural description (Clements, et al., 2002). A survey in 2006 (Lange et al., 2006) showed that, when the UML was used, it was mostly applied in a loose and informal way. The authors of that paper argued that this was a bad thing. I disagree with this view. The UML was designed for describing object-oriented systems and, at the architectural design stage, you often want to describe systems at a higher level of abstraction. Object classes are too close to the implementation to be useful for architectural description.

I don't find the UML to be useful during the design process itself and prefer informal notations that are quicker to write and which can be easily drawn on a whiteboard. The UML is of most value when you are documenting an architecture in detail or using model-driven development, as discussed in Chapter 5.

A number of researchers have proposed the use of more specialized architectural description languages (ADLs) (Bass et al., 2003) to describe system architectures. The basic elements of ADLs are components and connectors, and they include rules and guidelines for well-formed architectures. However, because of their specialized nature, domain and application specialists find it hard to understand and use ADLs. This makes it difficult to assess their usefulness for practical software engineering. ADLs designed for a particular domain (e.g., automobile systems) may be used as a

Name	MVC (Model-View-Controller)
Description	Separates presentation and interaction from the system data. The system is structured into three logical components that interact with each other. The Model component manages the system data and associated operations on that data. The View component defines and manages how the data is presented to the user. The Controller component manages user interaction (e.g., key presses, mouse clicks, etc.) and passes these interactions to the View and the Model. See Figure 6.3.
Example	Figure 6.4 shows the architecture of a web-based application system organized using the MVC pattern.
When used	Used when there are multiple ways to view and interact with data. Also used when the future requirements for interaction and presentation of data are unknown.
Advantages	Allows the data to change independently of its representation and vice versa. Supports presentation of the same data in different ways with changes made in one representation shown in all of them.
Disadvantages	Can involve additional code and code complexity when the data model and interactions are simple.

Figure 6.2 The model-view-controller (MVC) pattern

basis for model-driven development. However, I believe that informal models and notations, such as the UML, will remain the most commonly used ways of documenting system architectures.

Users of agile methods claim that detailed design documentation is mostly unused. It is, therefore, a waste of time and money to develop it. I largely agree with this view and I think that, for most systems, it is not worth developing a detailed architectural description from these four perspectives. You should develop the views that are useful for communication and not worry about whether or not your architectural documentation is complete. However, an exception to this is when you are developing critical systems, when you need to make a detailed dependability analysis of the system. You may need to convince external regulators that your system conforms to their regulations and so complete architectural documentation may be required.

6.3 Architectural patterns

The idea of patterns as a way of presenting, sharing, and reusing knowledge about software systems is now widely used. The trigger for this was the publication of a book on object-oriented design patterns (Gamma et al., 1995), which prompted the development of other types of pattern, such as patterns for organizational design (Coplien and Harrison, 2004), usability patterns (Usability Group, 1998), interaction (Martin and Sommerville, 2004), configuration management (Berczuk and