# 12. Object-oriented Design

## Objectives

The objective of this chapter is to introduce an approach to software design where the design is structured as interacting objects. When you have read this chapter, you will:

- understand how a software design may be represented as a set of interacting objects that manage their own state and operations,
- know the most important activities in a general object-oriented design process,
- understand different models that may be used to document an object-oriented design ,
- have been introduced to the representation of these models in the Unified Modeling Language (UML).

## Contents

12.1 **Objects and object classes**
12.2 **An object-oriented design process**
12.3 **Design evolution**

©Ian Sommerville 2000

Object-oriented design is a design strategy where system designers think in terms of 'things' instead of operations or functions. The executing system is made up of interacting objects that maintain their own local state and provide operations on that state information (Figure 12.1). They hide information about the representation of the state and hence limit access to it. An object-oriented design process involves designing the object classes and the relationships between these classes. When the design is realised as an executing program, the required objects are created dynamically using the class definitions.

Object-oriented design is part of *object-oriented development* where an object-oriented strategy is used throughout the development process:

- *Object-oriented analysis* is concerned with developing an object-oriented model of the application domain. The identified objects reflect entities and operations that are associated with the problem to be solved.

- *Object-oriented design* is concerned with developing an object-oriented model of a software system to implement the identified requirements. The objects in an object-oriented design are related to the solution to the problem that is being solved. There may be close relationships between some problem objects and some solution objects but the designer inevitably has to add new objects and to transform problem objects to implement the solution.

- *Object-oriented programming* is concerned with realising a software design using an object-oriented programming language. An object-oriented programming language, such as Java, supports the direct implementation of objects and provides facilities to define object classes.

The transition between these stages of development should be a seamless one with the same notation used at each stage. Moving to the next stage involves refining the previous stage by adding detail to existing object classes and devising new classes to provide additional functionality. As information is concealed within objects, detailed design decisions about the representation of data can be delayed until the system is implemented. In some cases, decisions on the distribution of objects and whether or not objects can be sequential or concurrent may also be delayed. This means that software designers are not constrained by details of the system implementation. They can devise designs that can be adapted to different execution environments.

Object-oriented systems should be maintainable as the objects are independent. They may be understood and modified as stand-alone entities. Changing the implementation of an object or adding services should not affect other system objects. Because objects are associated with things, there is often a clear mapping between real-world entities (such as hardware components) and their controlling objects in the system. This improves the understandability and hence the maintainability of the design.

Objects are potentially reusable components because they are independent encapsulations of state and operations. Designs can be developed using objects that have been created in previous designs. This reduces design, programming and validation costs. It may also lead to the use of standard objects (hence improving design understandability) and reduces the risks involved in software development.

©Ian Sommerville 2000

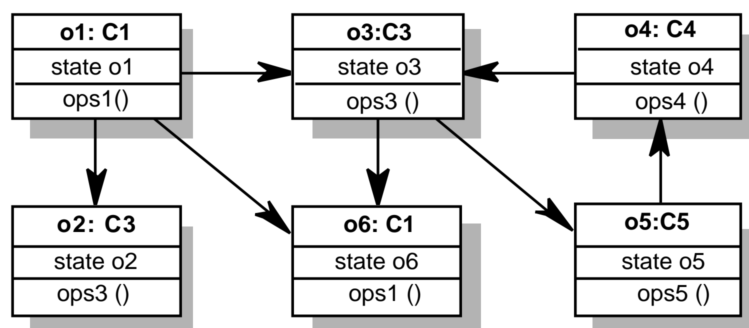


**Figure 12.1** A system made up of interacting objects

However, as I discuss in Chapter 14, reuse is sometimes best implemented using collections of objects (components or frameworks) rather than individual objects.

Several object-oriented design methods have been proposed (Coad and Yourdon, 1990; Robinson, 1992; Jacobson, Christensen et al., 1993; Booch, 1994; Graham, 1994). A unification of the notations used in these methods has been defined (UML) along with an associated design process (Rumbaugh, Jacobson et al., 1999). I don't describe any particular design method here but I discuss generic object-oriented concepts and design activities in section 12.2. I use the UML notation throughout the chapter.

## 12.1 Objects and object classes

The terms 'object' and 'object-oriented' are now widely used. They are applied to different types of entity, design methods, systems and programming languages. However, there is a general acceptance that an object is an encapsulation of information and this is reflected in my definition of an object and an object class:

> An object is an entity that has a state and a defined set of operations which operate on that state. The state is represented as a set of object attributes. The operations associated with the object provide services to other objects (clients) which request these services when some computation is required.
>
> Objects are created according to an object class definition. An object class definition serves as a template for creating objects. It includes declarations of all the attributes and operations which should be associated with an object of that class.

The notation that I use here for object classes is that defined in the UML. An object class is represented as a named rectangle with two sections. The object attributes are listed in the top section. The operations that are associated with the object are set out in the bottom section. Figure 12.2 illustrates this notation using an object class which models an employee in an organisation. In the UML, the term 'operation' is the specification of an action; the term 'method' is used to refer to the implementation of the operation.

©Ian Sommerville 2000

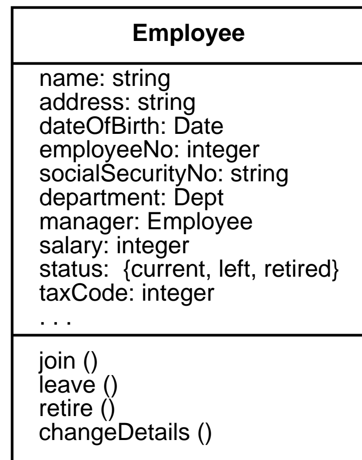


**Figure 12.2** An employee object

The class Employee defines a number of attributes that hold information about employees including their name and address, social security number, tax code, etc. The ellipsis (...) indicates that there are more attributes associated with the class that are not shown here. Operations associated with the object are join (called when an employee joins the organisation), leave (called when an employee leaves the organisation), retire (called when the employee becomes a pensioner of the organisation) and changeDetails (called when some employee information needs to be modified).

Objects communicate by requesting services (calling methods) from other objects and, if necessary, by exchanging the information required for service provision. The copies of information needed to execute the service and the results of service execution are passed as parameters. Some examples of this style of communication are:

```
// Call a method associated with a buffer object that returns the next value
// in the buffer
v = circularBuffer.Get () ;
// Call the method associated with a thermostat object that sets the
// temperature to be maintained
thermostat.setTemp (20) ;
```

In some distributed systems, object communications are implemented directly as text messages which objects exchange. The receiving object parses the message, identifies the service and the associated data and carries out the requested service. However, when the objects co-exist in the same program, method calls are implemented in the same way as procedure or function calls in a language such as C or Ada.

When service requests are implemented in this way, communication between objects is synchronous. That is, the calling object waits for the service request to be completed. However, if objects are implemented as concurrent processes or threads the object communication may be asynchronous. The calling object may continue in operation while the requested service is executing. I explain how objects may be implemented as concurrent processes later in this section.

©Ian Sommerville 2000

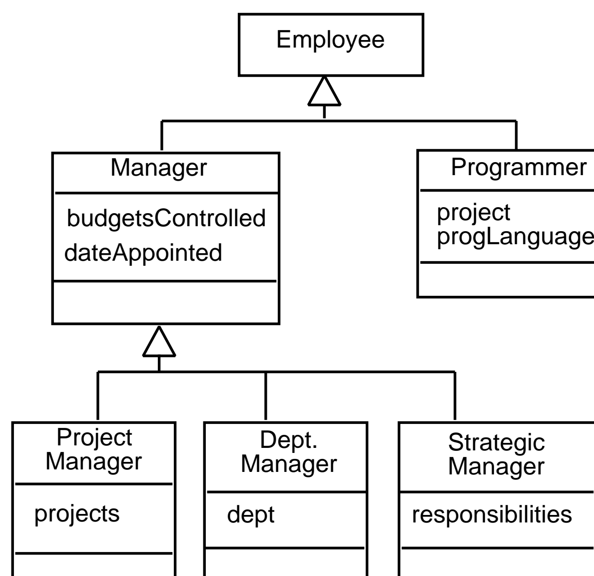


**Figure 12.3** A generalisation hierarchy

As I discussed in Chapter 7, where I described a number of possible object models, object classes can be arranged in a generalisation or inheritance hierarchy that shows the relationship between general and more specific object classes. The more specific object class is completely consistent with the general object class but includes further information. In the UML, generalisation is indicated by an arrow that points to the parent class. In object-oriented programming languages, generalisation is usually implemented using the inheritance mechanism. The child class inherits attributes and operations from the parent class.

Figure 12.3 shows an example of such a hierarchy where different classes of employee are shown. Classes lower down the hierarchy have the same attributes and operations as their parent classes but may add new attributes and operations or modify some of those from their parent classes. This means that there is one-way interchangability. If the name of a parent class is used in a model, this means that the object in the system may either be defined as that class or of any of its descendants.

The class Manager in Figure 12.3 has all of the attributes and operations of the class Employee but has, in addition, two new attributes that record the budgets controlled by the manager and the date that the manager was appointed to a particular management role. Similarly, the class Programmer adds new attributes that define the project that the programmer is working on and the programming language skills that they have. Objects of class Manager or Programmer may therefore be used anywhere where an object of class Employee is required.

Objects that are members of an object class participate in relationships with other objects. These relationships may be modelled by describing the associations between the object classes. In the UML, associations are denoted by a line between the object classes that may optionally be annotated with information about the association. This is illustrated in Figure 12.4 which shows the association between

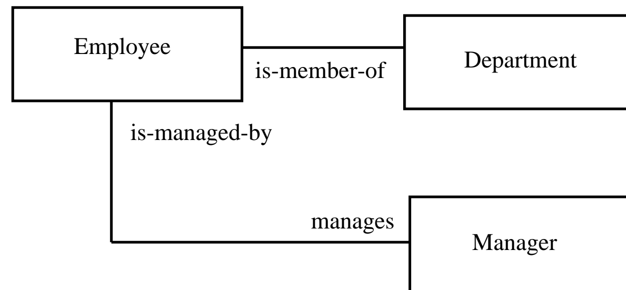


**Figure 12.4** An association model

objects of class Employee and objects of class Department and between objects of class Employee and objects of class Manager.

Association is a very general relationship and is often used in the UML to indicate that either an attribute of an object is an associated object or the implementation of an object method relies on the associated object. However, in principle at least, any kind of association is possible. One of the most common associations is aggregation which illustrates how objects may be composed of other objects. See Chapter 7 for a discussion of this type of association.

### 12.1.1 Concurrent objects

Conceptually, an object requests a service from another object by sending a 'service request' message to that object. There is no requirement for serial execution where one object waits for completion of a requested service. Consequently, the general model of object interaction allows objects to execute concurrently as parallel processes. These objects may execute on the same computer or as distributed objects on different machines.

In practice, most object-oriented programming languages have as their default a serial execution model where requests for object services are implemented in the same way as function calls. Therefore, when an object called theList is created from a normal object class, you write in Java:

```
theList.append (17)
```

This calls the append method associated with theList object to add the element 17 to theList and execution of the calling object is suspended until the append operation has been completed. However, Java includes a very simple mechanism (threads) that lets you create objects that execute concurrently. It is therefore easy to take an object-oriented design and produce an implementation where the objects are concurrent processes.

There are two kinds of concurrent object implementation:

1. *Servers* where the object is realised as a parallel process with methods corresponding to the defined object operations. Methods start up in response to an external message and may execute in parallel with methods associated with other objects. When they have completed their operation, the object suspends itself and waits for further requests for service.

```
class Transponder extends Thread {

        Position currentPosition ;
        Coords c1, c2 ;
        Satellite sat1, sat2 ;
        Navigator theNavigator ;

        public Position givePosition ()
        {
            return currentPosition ;
        }

        public void run ()
        {
            while (true)
            {
                c1 = sat1.position () ;
                c2 = sat2.position () ;
                currentPosition = theNavigator.compute (c1, c2) ;
            }

        }

} //Transponder
```

**Figure 12.5** Implementation of an active object using Java threads

2. *Active objects* where the state of the object may be changed by internal operations executing within the object itself. The process representing the object continually executes these operations so never suspends itself.

Servers are most useful in a distributed environment where the calling object and the called object execute on different computers. The response time for the service that is requested is unpredictable so, wherever possible, you should design the system so that the object that has requested a service does not have to wait for that service to be completed. They can also be used in a single machine where a service takes some time to complete (e.g. printing a document) and the service may be requested by several different objects.

Active objects are used when an object needs to update its own state at specified intervals. This is common in real-time systems where objects are associated with hardware devices that collect information about the system's environment. The object's methods allow other objects access to the state information.

Figure 12.5 shows how an active object may be defined and implemented in Java. This object class represents a transponder on an aircraft. The transponder keeps track of the aircraft's position using a satellite navigation system. It can respond to messages from air traffic control computers. It provides the current aircraft position in response to a request to the givePosition method.

This object is implemented as a thread where a continuous loop in the run method includes code to compute the aircraft's position using signals from satellites. Threads are created in Java by using the built-in Thread class as a parent class in a class declaration. Threads must include a method called run and this is

©Ian Sommerville 2000

started by the Java run-time system when objects that are defined as threads are created. This is shown in Figure 12.5.

## 12.2 An object-oriented design process

In this section, I illustrate the process of object-oriented design by developing an example design for the control software that is embedded in an automated weather station. As I discussed in the introduction, there are several methods of object-oriented design with no definitive 'best' method or design process. The process that I cover here is a general one that incorporates activities that are common to most OOD processes. In this respect, it is comparable to the proposed UML process (Rumbaugh, Jacobson et al., 1999) but I have significantly simplified this process for presentation here.

The general process that I use here for object-oriented design has a number of stages:

1. Understand and define the context and the modes of use of the system
2. Design the system architecture
3. Identify the principal objects in the system
4. Develop design models
5. Specify object interfaces

I have deliberately not illustrated this as a simple process diagram as that would imply that there was a neat sequence of activities in this process. In fact, all of the above activities can be thought of as inter-leaved activities that influence each other. Objects are identified and the interfaces fully or partially specified as the architecture of the system is defined. As object models are produced, these individual object definitions may be refined and this may mean changes to the system architecture.

I discuss these as separate stages in the design process later in this section. However, you should not assume from this that design is a simple, well-structured process. In reality, you develop a design by proposing solutions and refining these solutions as information becomes available. You inevitably have to backtrack and retry when problems arise. Sometimes you explore options in detail to see if they work; at other times you ignore details until late in the process.

I illustrate these process activities by developing an example of an object-oriented design. The example that I use to illustrate object-oriented design is part of a system for creating weather maps using automatically collected meteorological data. The detailed requirements for such a weather mapping system would take up many pages. However, an overall system architecture can be developed from a relatively brief system description:

> A weather mapping system is required to generate weather maps on a regular basis using data collected from remote, unattended weather stations and other data sources such as weather observers, balloons and satellites. Weather stations transmit their data to the area computer in response to a request from that machine.

©Ian Sommerville 2000

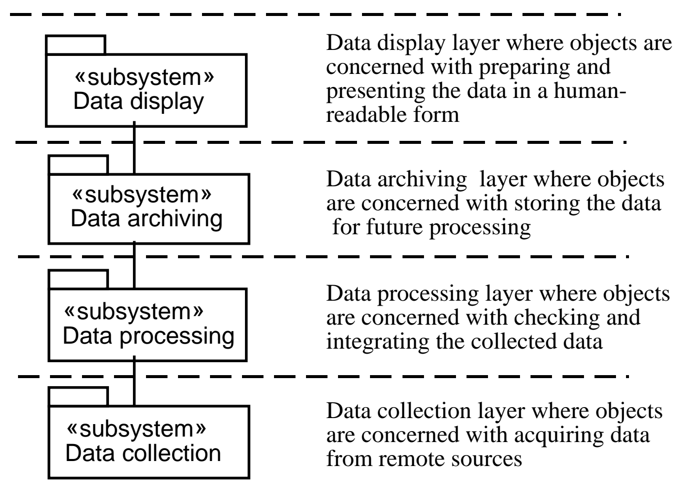


**Figure 12.6** Layered architecture for weather mapping system

> The area computer system validates the collected data and integrates the data from different sources. The integrated data is archived and, using data from this archive and a digitised map database, a set of local weather maps is created. Maps may be printed for distribution on a special-purpose map printer or may be displayed in a number of different formats.

This description shows that part of the overall system is concerned with collecting data, part with integrating the data from different sources, part with archiving that data and part with creating weather maps. Figure 12.6 illustrates a possible system architecture that can be derived from this description. This is a layered architecture (discussed in Chapter 10) that reflects the different stages of processing in the system namely data collection, data integration, data archiving and map generation. A layered architecture is appropriate in this case because each stage only relies on the processing of the previous stage for its operation.

In Figure 12.6, I have shown the different layers and have included the layer name in a UML package symbol that has been denoted as a subsystem. A UML package represents a collection of objects and other packages. I have used it here to show that each layer includes a number of other components.

In Figure 12.7 I have expanded on this abstract architectural model by showing that the components of the subsystems. Again, these are very abstract and they have been derived from the information in the description of the system. I continue the design example by focusing on the weather station subsystem that is part of the data collection layer.

### 12.2.1 System context and models of use

The first stage in any software design process is to develop an understanding of the relationships between the software that is being designed and its external environment. Developing this understanding helps you decide how to provide the required system functionality and how to structure the system so that it can communicate effectively with its environment.

©Ian Sommerville 2000

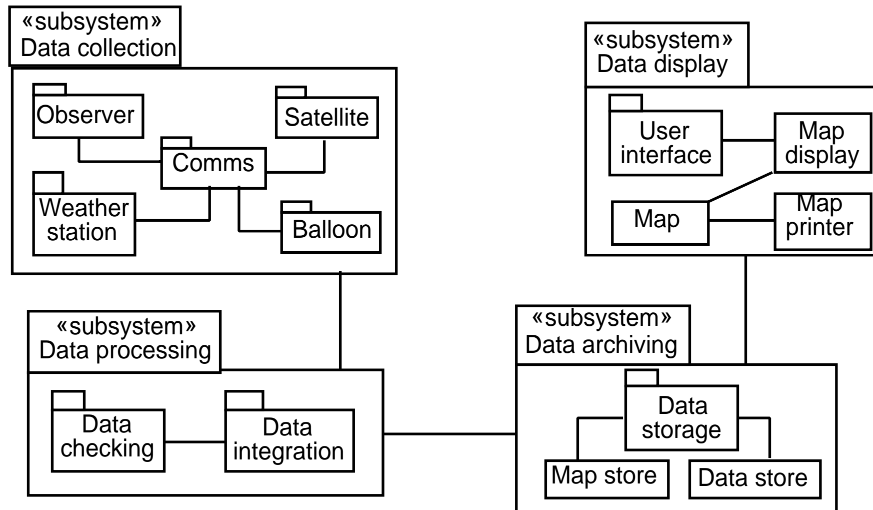


**Figure 12.7** Subsystems in the weather mapping system

The system context and the model of system use represent two complementary models of the relationships between a system and its environment:

1. The system context is a static model that describes the other systems in that environment.
2. The model of the system use is a dynamic model that describes how the system actually interacts with its environment.

The context model of a system may be represented using associations (see Figure 12.4) where, essentially, a simple block diagram of the overall system architecture is produced. This can be expanded by representing a subsystem model using UML packages as shown in Figure 12.7. This illustrates that the context of the weather station system is within a subsystem concerned with data collection. It also shows other subsystems that make up the weather mapping system.

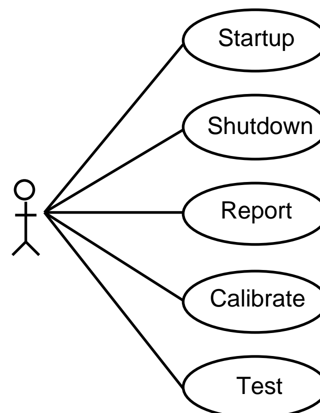


**Figure 12.8** Use-cases for the weather station

| System | Weather station |
|---|---|
| **Use-case** | Report |
| **Actors** | Weather data collection system, Weather station |
| **Data** | The weather station sends a summary of the weather data that has been collected from the instruments in the collection period to the weather data collection system. The data sent are the maximum minimum and average ground and air temperatures, the maximum, minimum and average air pressures, the maximum, minimum and average wind speeds, the total rainfall and the wind direction as sampled at 5 minute intervals. |
| **Stimulus** | The weather data collection system establishes a modem link with the weather station and requests transmission of the data. |
| **Response** | The summarised data is sent to the weather data collection system |
| **Comments** | Weather stations are usually asked to report once per hour but this frequency may differ from one station to the other and may be modified in future. |

**Figure 12.9** Report use-case description

When you model the interactions of a system with its environment you should use an abstract approach that does not include too much detail of these interactions. The approach that is proposed in the UML is to develop a use-case model where each use-case represents an interaction with the system. In use-case models (also discussed in Chapter 6), each possible interaction is named in an ellipse and the external entity involved in the interaction is represented by a stick figure. In the case of the weather station system, this external entity is not a human but the data processing system for the weather data.

A use-case model for the weather station is shown in Figure 12.8. This shows that weather station interacts with external entities for startup and shutdown, for reporting the weather data that has been collected and for instrument testing and calibration.

Each of these use cases can be described using a simple natural language description. This helps designers identify objects in the system and gives them an understanding of what the system is intended to do. I use a stylised form of this description that clearly identifies what information is exchanged, how the interaction is initiated etc. This is shown in Figure 12.9 where I have described the Report use case from Figure 12.8.

Of course, you can use any technique for describing use-cases so long as the description is short and easily understandable. You need to develop descriptions for each of the use-cases that are shown in the model.

The use-case description helps to identify objects and operations in the system. From the description of the Report use-case, it is obvious that objects representing the instruments that collect weather data will be required as will an object representing the summary of the weather data. Operations to request weather data and to send weather data are required.

### 12.2.2 Architectural design

Once the interactions between the software system that is being designed and the system's environment have been defined, you can then use this information as a

©Ian Sommerville 2000

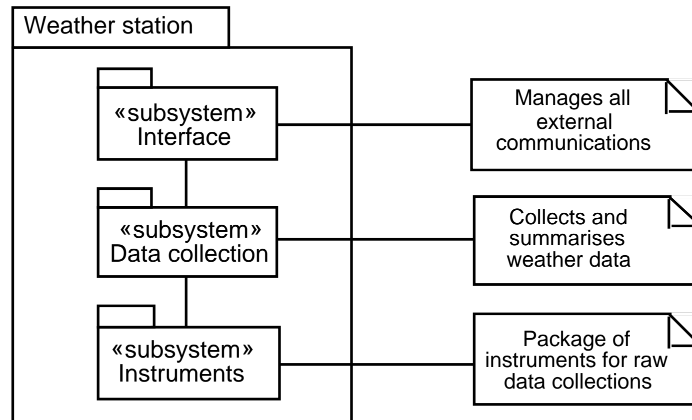


**Figure 12.10** The weather station architecture

basis for designing the system architecture. Of course, you need to combine this with your general knowledge of principles of architectural design and with more detailed domain knowledge.

The automated weather station is a relatively simple system and its architecture can again be represented as a layered model. I have illustrated this in Figure 12.10 as three UML packages within the more general Weather station package. Notice how I have used UML annotations (text in boxes with a folded corner) to provide additional information here.

The three layers in the weather station software are:

1. The interface layer which is concerned with all communications with other parts of the system and with providing the external interfaces of the system.

2. The data collection layer which is concerned with managing the collection of data from the instruments and with summarising the weather data before transmission to the mapping system.

3. The instruments layer which is an encapsulation of all of the instruments that are used to collect raw data about the weather conditions.

In general, you should try and decompose a system so that architectures are as simple as possible. A good rule of thumb is that there should not be more than seven fundamental entities included in an architectural model. Each of these entities can be described separately but, of course, you may choose to reveal the structure of the entities as I have done in Figure 12.7.

### 12.2.3 Object identification

By this stage in the design process, you will already have formulated some ideas about the essential objects in the system that you are designing. In the weather station system, it is clear that the instruments are objects and you need at least one object at each of the architectural levels. This reflects a general principle that objects tend to emerge during the design process. However, you usually also have to look for and document other objects that may be relevant.

©Ian Sommerville 2000

Although I have headed this section *object* identification, in practice this process is actually concerned with identifying *object classes*. The design is described in terms of these classes. Inevitably, you have to refine the object classes that you initially identify and revisit this stage of the process as you develop a deeper understanding of the design.

There have been various proposals made about how to identify objects classes:

1. Use a grammatical analysis of a natural language description of a system. Objects and attributes are nouns, operations or services are verbs. (Abbott, 1983). This approach has been embodied in the HOOD method for object-oriented design (Robinson, 1992) that has been widely used in the European aerospace industry.

2. Use tangible entities (things) in the application domain such as aircraft, roles such as manager, events such as request, interactions such as meetings, locations such as offices, organisational units such as companies, etc. (Shlaer and Mellor, 1988; Coad and Yourdon, 1990; Wirfs-Brock, Wilkerson et al., 1990).Support this by identifying storage structures (abstract data structures) in the solution domain that might be required to support these objects.

3. Use a behavioural approach where the designer first understands the overall behaviour of the system. The various behaviours are assigned to different parts of the system and an understanding is derived of who initiates and participates in these behaviours. Participants who play significant roles are recognised as objects (Rubin and Goldberg, 1992).

4. Use a scenario-based analysis where various scenarios of system use are identified and analysed in turn. As each scenario is analysed, the team responsible for the analysis must identify the required objects, attributes and operations. A method of analysis called CRC cards where analysts and designers take on the role of objects is effective in supporting this scenario-based approach (Beck and Cunningham, 1989).

These approaches help you get started with object identification. In practice, many different sources of knowledge have to be used to discover objects and object classes. Objects and operations that are initially identified from the informal system description can be a starting point for the design. Further information from application domain knowledge or scenario analysis may then be used to refine and extend the initial objects. This information may be collected from requirements documents, from discussions with users and from an analysis of existing systems.

I have used a hybrid approach here to identify the weather station objects. I don't have space to describe all the objects but I have shown five object classes in Figure 12.11. The Ground thermometer, Anemometer and Barometer represent application domain objects and the WeatherStation and WeatherData objects have been identified from the system description and the scenario (use-case) description.

These objects are related to the different levels in the system architecture.

1. The WeatherStation object class provides the basic interface of the weather station with its environment. Its operations therefore reflect the interactions

©Ian Sommerville 2000

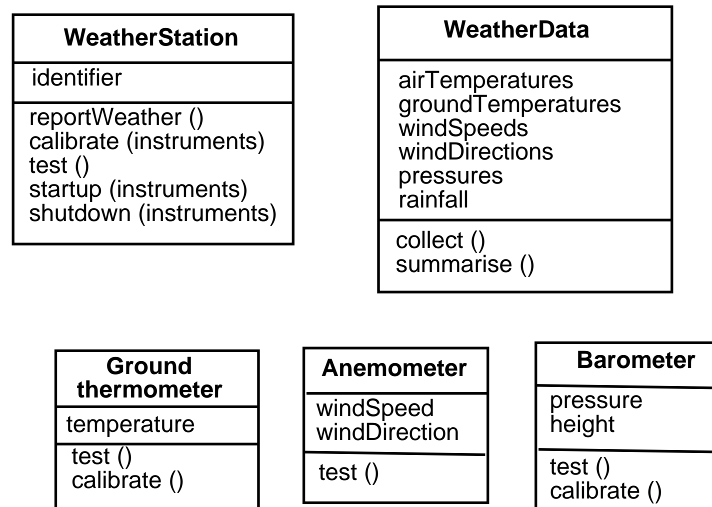


**Figure 12.11** Examples of object classes in the weather station system

shown in Figure 12.8. In this case, I use a single object class to encapsulate all of these interactions but, in other designs, it may be more appropriate to use several classes to provide the system interface.

2. The WeatherData object class encapsulates the summarised data from the different instruments in the weather station. Its associated operations are concerned with collecting and summarising the data that is required.

3. The GroundThermometer, Anemometer and Barometer object classes are directly related to instruments in the system. They reflect tangible hardware entities in the system and the operations are concerned with controlling that hardware.

At this stage in the design process, knowledge of the application domain may be used to identify further objects and services. In this case, we know that weather stations are often located in remote places. They include various instruments which sometimes go wrong. Instrument failures should be reported automatically. This implies that attributes and operations to check the correct functioning of the instruments are necessary. Obviously, there are many remote weather stations. You therefore need some way of identifying the data collected from each station so each weather station should have its own identifier.

I have decided not to make the objects associated with each instrument active objects. The collect operation in WeatherData calls on instrument objects to make readings when required. Active objects includes their own control and, in this case, it would mean that each instrument would decide when to make readings. However, the disadvantage of this is that, if a decision was made to change the timing of the data collection or if different weather stations collected data differently then new object classes would have to be introduced. By making the instrument objects make readings on request, any changes to collection strategy can be easily implemented without changing the objects associated with the instruments.

©Ian Sommerville 2000

### 12.2.4 Design models

Design models show the objects or object classes in a system and, where appropriate, different kinds of relationships between these entities. Design models essentially are the design. They are the bridge between the requirements for the system and the system implementation. This means that there are conflicting requirements on these models. They have to be abstract so that unnecessary detail doesn't hide the relationships between them and the system requirements. However, they also have to include enough detail for programmers to make implementation decisions.

In general, you get round this conflict by developing different models at different levels of detail. Where there are close links between requirements engineers, designers and programmers then abstract models may be all that are required. Specific design decisions may be made as the system is implemented. When the links between system specifiers, designers and programmers are indirect (e.g. where a system is being designed in one part of an organisation but implemented elsewhere) then more detailed models may be required.

An important step in the design process, therefore, is to decide which design models that you need and the level of detail of these models. This also depends on the type of system that is being developed. A sequential data processing system will be designed in a different way from an embedded real-time system and different design models will therefore be used. There are very few systems where all models are necessary and minimising the number of models that are produced reduces the costs of the design and the time required to complete the design process.

There are two types of design models that should normally be produced to describe an object-oriented design. These are:

1. Static models that describe the static structure of the system in terms of the system object classes and their relationships. Important relationships that may be documented at this stage are generalisation relationships, uses/used-by relationships and composition relationships.

2. Dynamic models that describe the dynamic structure of the system and that show the interactions between the system objects (not the object classes). Interactions that may be documented include the sequence of service requests made by objects and the way in which the state of the system is related to these object interactions.

The UML provides for a large number of possible static and dynamic models that may be produced to document a design. Booch *et al.* (Booch, Rumbaugh et al., 1999) propose 9 different types of diagram to represent these models. I don't have space to go into all of these and not all are appropriate for the weather station example. The models that I will discuss in this section are:

1. Subsystem models that show logical groupings of objects into coherent subsystems. These are represented using a form of class diagram where each subsystem is shown as a package. Subsystem models are static models.

2. Sequence models that show the sequence of object interactions. These are represented using a UML sequence or a collaboration diagram. Sequence models are dynamic models.

©Ian Sommerville 2000

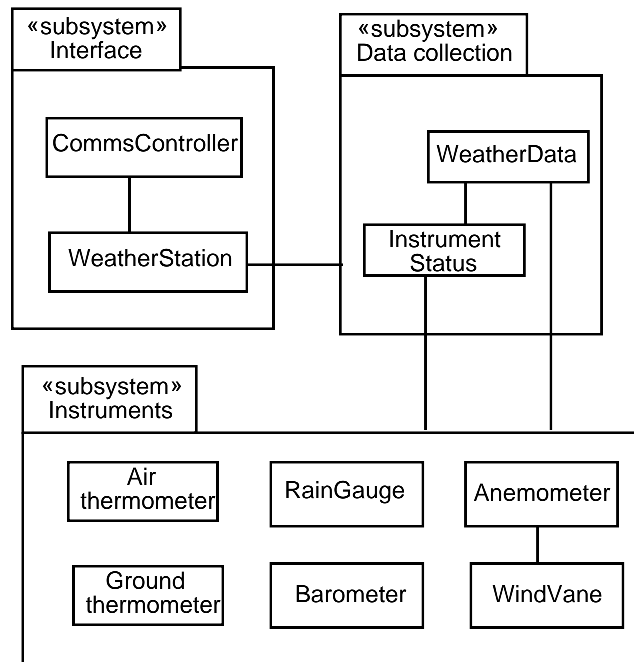


**Figure 12.12** Weather station packages

2. State machine models that show how individual objects change their state in response to events. These are represented in the UML using statechart diagrams. State machine models are dynamic models.

I have already discussed other types of model that may be developed earlier in this chapter and in other chapters. Use-case models show interactions with the system (Figure 12.8, Figures 6.12 and 6.13, Chapter 6), object models describe the object classes (Figure 12.2), generalisation or inheritance models (Figure 7.10, Figure 7.11 and Figure 7.12, Chapter 7) show how classes may be generalisations of other classes and aggregation models (Figure 7.13) show how collections of objects are related.

A subsystem model is, in my view, one of the most helpful static models as it shows how the design may be organised into logically related groups of objects. We have already seen examples of this type of model in Figure 12.7 that showed the subsystems in the weather mapping system. In the UML packages are encapsulation constructs and do not reflect directly on entities in the system that is developed. However, they may be reflected in structuring constructs such as Java libraries.

Figure 12.12 shows the objects in the subsystems in the weather station. I also show some associations in this model. For example, the CommsController object is associated with the WeatherStation object and the WeatherStation object is associated with the Data collection package. This means that this object is associated with one or more objects in this package. A package model plus an object class model should describe the logical groupings in the system.

©Ian Sommerville 2000

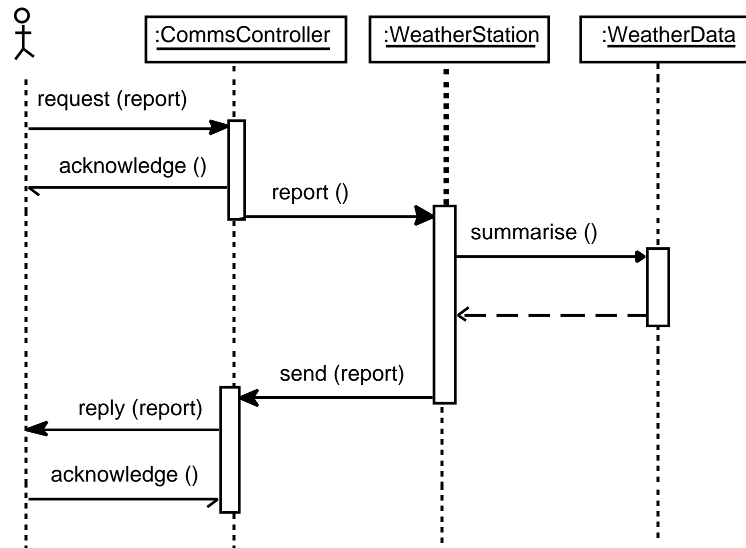


**Figure 12.13**
Sequence of operations - data collection

One of the most useful and understandable dynamic models that may be produced is a sequence model that documents, for each mode of interaction, the sequence of object interactions that take place. In a sequence model:

1. The objects involved in the interaction are arranged horizontally with a vertical line linked to each object.
2. Time is represented vertically so that time progresses down the dashed vertical lines. Therefore, the sequence of operations can be read easily from the model.
3. Interactions between objects are represented by labelled arrows linking the vertical lines. These are *not* data flows but represent messages or events that are fundamental to the interaction.
4. The thin rectangle on the object lifeline represents the time when the object is the controlling object in the system. An object takes over control at the top of this rectangle and relinquishes control to another object at the bottom of the rectangle. If there is a hierarchy of calls, control is not relinquished until the last return to the initial method call has been completed.

This is illustrated in Figure 12.13 that shows the sequence of interactions when the external mapping system requests the data from the weather station. This diagram may be read as follows:

1. An object that is an instance of CommsController (:CommsController) receives a request from its environment to send a weather report. It acknowledges receipt of this request. The half-arrowhead indicates that the message sender does not expect a reply.
2. This object sends a message to an object that is an instance of WeatherStation to create a weather report. The instance of CommsController

©Ian Sommerville 2000

then suspends itself (its control box ends). The style of arrowhead used indicates that the CommsController object instance and the WeatherStation object instance are objects that may execute concurrently.

3. The object that is an instance of WeatherStation sends a message to a WeatherData object to summarise the weather data. In this case, the different style of arrowhead used indicates that the instance of WeatherStation waits for a reply.

4. This summary is computed and control returns to the WeatherStation object. The dotted arrow indicates a return of control.

5. This object sends a message to CommsController requesting it to transfer the data to the remote system. The WeatherStation object then suspends itself.

6. The CommsController object sends the summarised data to the remote system, receives an acknowledgement and then suspends itself waiting for the next request.

From the sequence diagram we can see that the CommsController object and the WeatherStation object are actually concurrent processes where execution can be suspended and resumed. Essentially, the CommsController object instance listens for messages from the external system, decodes these messages and initiates weather station operations.

When documenting a design, you should produce a sequence diagram for each significant interaction. If you have developed a use-case model then there should be a sequence diagram for each use-case that you have identified.

Sequence diagrams are used to model the combined behaviour of a group of objects but you may also want to summarise the behaviour of a single object in response to the different messages that it can process. To do this, you can use a state machine model that shows how the object instance changes state depending on the messages that it receives. The UML uses statecharts, initially invented by Harel (Harel, 1987) to describe state machine models.

Figure 12.14 is a statechart for the WeatherStation object that shows how it responds to requests for various services.

You can read this diagram as follows:

1. If the object state is 'Shutdown' then it can only respond to a startup () message. It then moves into a state where it is waiting for further messages. The unlabelled arrow with the black blob indicates that the 'Shutdown' state is the initial state.

2. In the 'Waiting' state, the system expects further messages. If a shutdown () message is received, the object returns to the shutdown state.

3. If a reportWeather () message is received, the system moves to a summarising state then, when the summary is complete, to a transmitting state where the information is transmitted through the CommsController. It then returns to a waiting state.

4. If a calibrate () message is received, the system moves to a calibrating state then a testing state then a transmitting state before returning to the waiting

©Ian Sommerville 2000

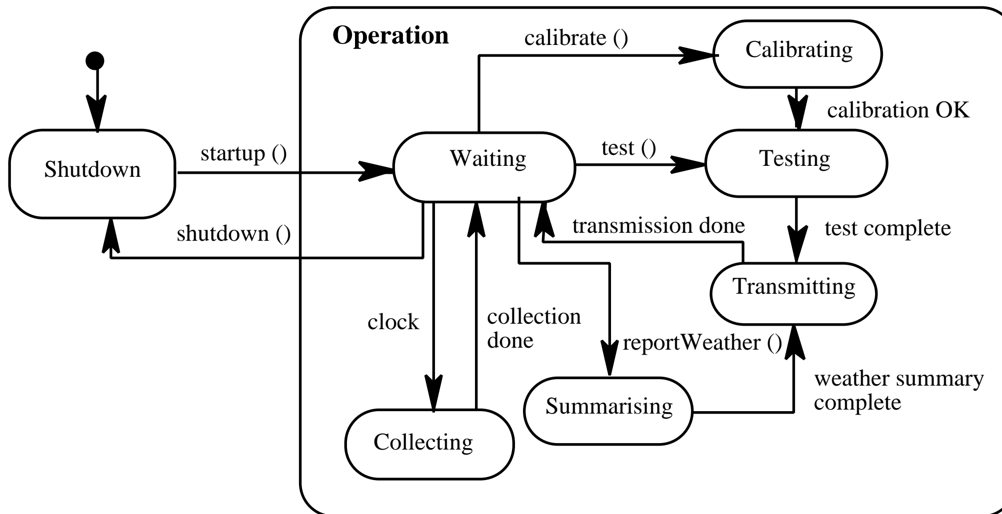


**Figure 12.14**
State diagram for WeatherStation

state. If a test () message is received, the system moves directly to the testing state.

2. If a signal from the clock is received, the system moves to a collecting state where it is collecting data from the instruments. Each instrument is instructed in turn to collect its data.

It is not usually necessary to produce a statechart for all of the objects that you have defined. Many of the objects in a system are relatively simple objects and a state machine model would not help implementors to understand these objects.

### 12.2.5 Object interface specification

An important part of any design process is the specification of the interfaces between the different components in the design. You need to specify interfaces so that objects and other components can be designed in parallel. Once an interface has been specified, the developers of other objects may assume that interface will be implemented.

Designers should avoid interface representation information in their interface design. Rather the representation should be hidden and object operations provided to access and update the data. If the representation is hidden, it can be changed without affecting the objects that use these attributes. This leads to a design which is inherently more maintainable. For example, an array representation of a stack may be changed to a list representation without affecting other objects which use the stack. By contrast, it often makes sense to expose the attributes in a static design model as this is the most compact way of illustrating essential characteristics of the objects.

There is not necessarily a simple 1:1 relationship between objects and interfaces. The same object may have several interfaces that are each viewpoints on the methods that it provides. This is supported directly in Java where interfaces are

```
interface WeatherStation {

    public void WeatherStation () ;

    public void startup () ;
    public void startup (Instrument i) ;

    public void shutdown () ;
    public void shutdown (Instrument i) ;

    public void reportWeather ( ) ;

    public void test () ;
    public void test ( Instrument i ) ;

    public void calibrate ( Instrument i) ;

    public int getID () ;

} //WeatherStation
```

**Figure 12.15** Java description of weather station interface

declared separately from objects and objects 'implement' interfaces. Equally, a group of objects may all be accessed through a single interface.

Object interface design is concerned with specifying the detail of the interface to an object or to a group of objects. This means defining the signatures and semantics of the services that are provided by the object or by a group of objects. Interfaces can be specified in the UML using the same notation as in a class diagram. However, there is no attribute section and the UML stereotype <<interface>> should be included in the name part.

An alternative approach that I prefer is to use a programming language to define the interface. This is illustrated in Figure 12.15 that shows the interface specification in Java of the weather station. As interfaces become more complex, this approach becomes more effective because the syntax checking facilities in the compiler may be used to discover errors and inconsistencies in the interface description. The Java description can show that some methods can take different numbers of parameters. Therefore, the shutdown method can either be applied to the station as a whole if it has no parameters or can shutdown a single instrument.

## 12.3 Design evolution

An important advantage of an object-oriented approach to design is that it simplifies the problem of making changes to the design. The reason for this is that object state representation does not influence the design. Changing the internal details of an object is unlikely to affect any other system objects. Furthermore, because objects are loosely coupled, it is usually straightforward to introduce new objects without significant effects on the rest of the system.

To illustrate the robustness of the object-oriented approach, assume that pollution monitoring capabilities are to be added to each weather station. This involves adding an air quality meter to compute the amount of various pollutants in

©Ian Sommerville 2000

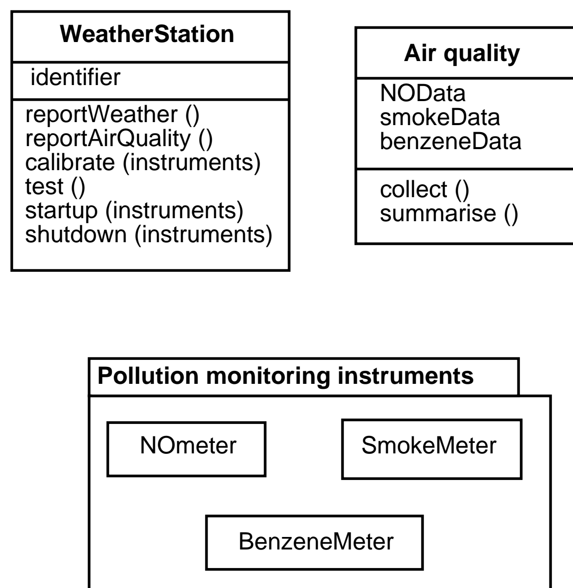


**Figure 12.16** New objects to support pollution monitoring

the atmosphere. The pollution readings are transmitted at the same time as the weather data. To modify the design, the following changes must be made:

1. An object class called Air quality should be introduced as part of WeatherStation at the same level as WeatherData.

2. An operation reportAirQuality should be added to Weather Station to send the pollution information to the central computer. The weather station control software must be modified so that pollution readings are automatically collected when requested by the top level WeatherStation object.

2. Objects representing the types of pollution monitoring instruments should be added. In this case, levels of nitrous oxide, smoke and benzene can be measured.

Figure 12.16 shows Weather station and the new objects added to the system. Apart from at the highest level of the system (WeatherStation) no software changes are required in the original objects in the weather station. The addition of pollution data collection does not affect weather data collection in any way.

## KEY POINTS

- Object-oriented design is a means of designing software so that the fundamental components in the design represent objects with their own private state and operations rather than functions.
- An object should have constructor and inspection operations allowing its state to be inspected and modified. The object provides services (operations using state information) to other objects. Objects are created at run-time using a specification in an object class definition.

- Objects may be implemented sequentially or concurrently. A concurrent object may be a passive object whose state is only changed through its interface or an active object that can change its own state without outside intervention.
- The Unified Modeling Language (UML) has been designed to provide a range of different notations that can be used to document an object-oriented design.
- The process of object-oriented design includes activities to design the system architecture, identify objects in the system, describe the design using different object models and document the object interfaces.
- A range of different models may be produced during an object-oriented design process. These include static models (class models, generalisation models, association models) and dynamic models (sequence models, state machine models)
- Object interfaces must be defined precisely so that they can be used by other objects. A programming language such as Java may be used to document object interfaces.
- An important advantage of object-oriented design is that it simplifies the evolution of the system.

## FURTHER READING

*Comm. ACM, October 1999.* This special issue includes a number of articles on the practical use of the UML for design. The article by Bell and Schmidt on using a design case study is particularly recommended.

*Using UML: Software Engineering with Objects and Components* A nice, easy-to-read introduction to object-oriented design using UML. (R. Pooley and P. Stevens, Addison-Wesley Longman, 1999)

*The Unified Modeling Language User Guide* The definitive text on UML and its use for describing object-oriented designs. There are 2 other associated texts - one is a UML reference manual, the other proposes an object-oriented development process (G. Booch, I. Jacobson and J. Rumbaugh, Addison-Wesley Longman, 1999)

## EXERCISES

12.1 Explain why adopting an approach to design that is based on loosely coupled objects that hide information about their representation should lead to a design which may be readily modified.

12.2 Using examples, explain the difference between an object and an object class.

12.3 Under what circumstances might it be appropriate to develop a design where objects execute concurrently?

12.4 Using the UML graphical notation for object classes, design the following object classes identifying attributes and operations. Use your own experience to decide on the attributes and operations that should be associated with these objects.

- A telephone
- A printer for a personal computer
- A personal stereo system
- A bank account
- A library catalogue

12.5 Develop the design of the weather station design in detail by proposing interface descriptions of the objects shown in Figure 12.11. This may be expressed in Java, in C++, or in the UML.

12.6 Develop the design of the weather station to show the interaction between the data collection subsystem and the instruments that collect weather data. Use sequence charts to show this interaction.

12.7 Identify possible objects in the following systems and develop an object-oriented design for them. You may make any reasonable assumptions about the systems when deriving the design.

- A group diary and time management system is intended to support the timetabling of meetings and appointments across a group of co-workers. When an appointment is to be made which involves a number of people, the system finds a common slot in each of their diaries and arranges the appointment for that time. If no common slots are available, it interacts with the user to rearrange their personal diary to make room for the appointment.
- A petrol (gas) station is to be set up for fully automated operation. Drivers swipe their credit card through a reader connected to the pump, the card is verified by communication with a credit company computer and a fuel limit established. The driver may then take the fuel required. When fuel delivery is complete and the pump hose is returned to its holster, the driver’s credit card account is debited with the cost of the fuel taken. The credit card is returned after debiting. If the card is invalid, it is returned by the pump before fuel is dispensed.

12.8 Write precise interface definitions in Java or C++ for the objects you have defined in Exercise 12.7.

12.9 Draw a sequence diagram showing the interactions of objects in a group diary system when a group of people arrange a meeting.

12.10 Draw a statechart showing the possible state changes in one or more of the objects that you have defined in Exercise 12.7.

©Ian Sommerville 2000