2

MODELLING WITH OBJECTS

Object-oriented programming and design languages share a common understanding of what software is and how programs work. The *object model* is the common computational model shared by UML and object-oriented programming languages. Programming and design languages express facts about programs at different levels of abstraction, but our understanding of both types of language is based on the abstract description of running programs that is provided by this model.

This chapter describes the essential features of the object model, introducing them in the context of a simple application. The notation provided by UML for illustrating these concepts is presented and the chapter also illustrates how they can be implemented, thus making clear the close connections between design and programming languages.

2.1 THE OBJECT MODEL

The object model is not a specific UML model, but a general way of thinking about the structure of programs. It consists of a framework of concepts that underlie object-oriented design and programming activity. As the name suggests, the fundamental property of the object model is that computation takes place in and between *objects*.

Individual objects are responsible for maintaining part of a system's data and for implementing aspects of its overall functionality. When a program is running, an object is typically represented by an area of memory containing, among other things, the data stored by that object. Objects also support methods, or functions, to access and update the data that they contain. Objects therefore combine two fundamental aspects of computer programs, namely data and processing, that are kept separate in other approaches to software design.

A program is more than a collection of isolated objects, however. Relationships between the data stored in individual objects must be recorded, and the global behaviour of the program only emerges from the interaction of many distinct objects. These requirements are supported by allowing objects to be linked together. This is typically achieved by enabling one object to hold a reference to another, or in more concrete terms, to know the location of the other object.

The object model therefore views a running program as a network, or graph, of objects. The objects form the nodes in the graph, and the arcs connecting the objects are known as *links*. Each object contains a small subset of the program's data, and the structure of the object network represents relationships between this data. Objects can be created and destroyed at run-time and the links between them can also be changed. The structure, or topology, of the object network is therefore highly dynamic, changing as a program runs.

The links between objects also serve as communication paths, which enable objects to interact by sending *messages* to each other. Messages are analogous to function calls: they are typically requests for the receiving object to perform one of its methods and can be accompanied by data parameterizing the message. Often, an object's response to a message will be to send messages to other objects and in this way computation can spread across the network, involving many objects in the response to an initial message.

It is possible to describe the structure of the object graph in a running program and to trace the effect of individual messages: a debugger would be a suitable tool for doing this. It is not normally feasible to write programs by defining individual objects, however. Instead, structural descriptions of *classes* of similar objects are given, defining the data that they can hold and the effect that execution of their methods has. The source code of an object-oriented program, therefore, does not directly describe the object graph, but rather the properties of the objects that make it up.

The role of the object model in design

In the context of design, the importance of the object model is that it provides a semantic foundation for UML's design notations. The meaning of many features in UML can be understood by interpreting them as statements about sets of connected, intercommunicating objects.

UML diagrams can be drawn to represent particular run-time configurations of objects. It is much more common, however, to draw diagrams that play the same role as source code, defining in a general structural way what can happen at run-time. These diagrams fall into two main categories. *Static* diagrams describe the kinds of connections that can exist between objects and the possible topologies that the resulting object network can have. *Dynamic* diagrams describe the messages that can be passed between objects and the effect on an object of receiving a message.

The dual role of the object model makes it very easy to relate UML design notations to actual programs, and this explains why UML is a suitable language for designing and documenting object-oriented programs. The remainder of this chapter illustrates this by using some basic UML notations to document a simple program.

A stock control example

In manufacturing environments, where complex products of some sort are being assembled out of component parts, a common requirement is to keep track of the stock of parts held and the way in which these parts are used. In this chapter we will illustrate the object model by developing a simple program that models different kinds of parts and their properties, and the ways in which these parts are used to construct complex assemblies.

The program will have to manage information describing the different parts known to the system. As well as maintaining information about the different types of part in use, we will assume that it is important for the system to keep track of individual physical parts, perhaps for the purposes of quality assurance and tracking.

For the purposes of this example, we will assume that we are interested in the following three pieces of information about each part.

1. Its catalogue reference number (an integer).
2. Its name (a string).
3. The cost of an individual part (a floating point value).

Parts can be assembled into more complex structures called *assemblies*. An assembly can contain any number of parts and can have a hierarchical structure. In other words, an assembly can be made up of a number of subassemblies, each of which in turn is composed of parts and possibly further subassemblies of its own.

A program which maintained information about parts, assemblies and their structure could be used for a number of purposes, such as maintaining catalogue and inventory information, recording the structure of manufactured assemblies and supporting various operations on assemblies, such as calculating the total cost of the parts in an assembly or printing a listing of all its parts. In this chapter we will consider one simple application, a query function that will find out the cost of the materials in an assembly by adding up the costs of all the parts it contains.

2.2 CLASSES AND OBJECTS

The data and functionality in an object-oriented system is distributed among the objects that exist while the system is running. Each individual object maintains part of the system's data and provides a set of methods that permit other objects in the system to perform certain operations on that data. One of the hard tasks of object-oriented design is deciding how to split up a system's data into a set of objects that will interact successfully to support the required overall functionality.

A frequently applied rule of thumb for the identification of objects is to represent real-world objects from the application domain by objects in the model. One of the major tasks of the stock control system is to keep track of all the physical parts held in stock by the manufacturer. A natural starting point, therefore, is to consider representing each of these parts as an object in the system.

In general there will be lots of part objects, each describing a different part and so storing different data, but each having the same structure. The common structure of a set of objects which represent the same kind of entity is described by a *class*, and each object of that kind is said to be an *instance* of the class. As a first step in the design of the stock management system, then, we might think of defining a 'part' class.

Once a candidate class has been identified, we can consider what data should be held by instances of that class. A natural suggestion in the case of parts is that each object should contain the information that the system must hold about that part: its name, number and cost. This suggestion is reflected in the following Java class.

```
public class Part
{
  private String name ;
  private long   number ;
  private double cost ;

  public Part(String nm, long num, double cst) {
    name   = nm ;
    number = num ;
    cost   = cst ;
  }

  public String getName()   { return name ; }
  public long   getNumber() { return number ; }
  public double getCost()   { return cost ; }
}
```

The UML notion of a class is very similar to that found in programming languages such as C++ and Java. A UML class defines a number of *features*: these are subdivided into *attributes*, which define the data stored by instances of the class, and *operations*, which define their behaviour. Generally speaking, attributes correspond to the fields of a Java class and operations to its methods.

In UML, classes are represented graphically by a rectangular icon divided into three compartments containing the name, attributes and operations of the class, respectively. The UML representation of the `Part` class above is shown in Figure 2.1.

The top section of the class icon contains the name of the class, the second section contains its attributes and the third section its operations. Programming language types can be used in operation signatures and a colon is used to separate the type from the name of an attribute, parameter or operation. UML also shows the access levels of the various features of the class, using a minus sign for 'private' and a plus sign for 'public'. Constructors are underlined to distinguish them from normal instance methods of the class.

Full details of the UML syntax are given in Chapter 8, but it is worth noting at this point that much of the detail shown in Figure 2.1 is optional. All that is required is the compartment containing the name of the class: other information can be omitted if it is not required in a particular diagram.

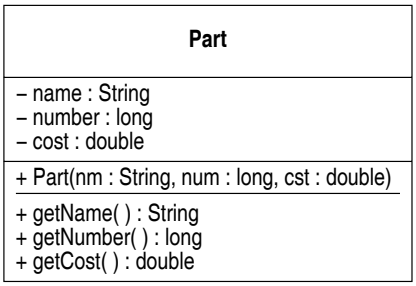

Figure 2.1 The `Part` class represented in UML

Object creation

Classes are defined at compile-time, but objects are created at run-time, as instances of classes. Execution of the following statement results in the creation of a new object. This involves two steps: an area of memory is first allocated for the object and then it is suitably initialized. Once the new object is created, a reference to it is stored in the variable `myScrew`.

```
Part myScrew = new Part("screw", 28834, 0.02) ;
```

Graphical notation for depicting individual objects and the data held in them is defined by UML. The object created by the line of code above could be depicted in UML as shown in Figure 2.2.

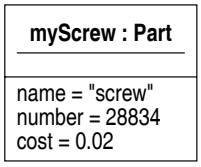

Figure 2.2 A part object

The object is represented by a rectangle divided into two compartments. The upper compartment contains the object's name followed by the name of its class, both underlined. Objects do not have to be named, but where a variable is associated with an object it is sometimes useful to use the variable name to name the object as well. An object's class is always shown when it is known. A common stylistic convention is to choose class names that start with an upper-case letter and object names that start with a lower-case letter.

Data is held in objects as the values of attributes. The lower compartment of an object icon contains the names and current values of the object's attributes. This compartment is optional, and can be omitted if it is not necessary to show the values of an object's attributes on a diagram.

2.3 OBJECT PROPERTIES

A common characterization of objects states that an object is something which has state, behaviour and identity. These notions are explained in more detail below, together with the related notion of encapsulation. The various terms are also related to those features of a class definition that implement them.

State

The first significant feature of objects is that they act as containers for data. In Figure 2.2, this property of objects is pictured by including the data inside the icon representing the object. In a pure object-oriented system all the data maintained by the system is stored in objects: there is no notion of global data or of a central data repository as there is in other models.

The data values contained in an object's attributes are often referred to as the object's *state*. For example, the three attribute values shown in Figure 2.2 comprise the state of the object 'myScrew'. As these data values will change as a system evolves, it follows that an object's state can change too. In object-oriented programming languages an object's state is specified by the fields defined in the object's class, and in UML by the attributes of the class. The three attribute values shown in Figure 2.2 therefore correspond to the three fields defined in the `Part` class defined in Section 2.2.

Behaviour

In addition to storing data, each object provides an *interface* consisting of a number of operations. Normally some of these operations will provide access and update functions for the data stored inside the object, but others will be more general and implement some aspect of the system's global functionality.

In programming languages, an object's operations are defined in its class, as a set of methods. The set of operations defined by an object defines that object's interface. For example, the interface of the part class defined in Section 2.2 consists of a constructor, and access functions to return the data stored in an object's fields.

Unlike attributes, operations are not shown on object icons in UML. The reason for this is that an object provides exactly the operations that are defined by its class. As a class can have many instances, each of which provides the same operations, it would be redundant to show the operations for each object. In this respect, an object's behaviour differs from its state as, in general, different instances of the same class will store different data, and hence have a different state.

Identity

A third aspect of the definition of objects is that every object is distinguishable from every other object. This is the case even if two objects contain exactly the same data and provide exactly the same set of operations in their interface. For example, the following lines of code create two objects that have the same state, but are nevertheless distinct.

```
Part screw1 = new Part("screw", 28834, 0.02) ;
Part screw2 = new Part("screw", 28834, 0.02) ;
```

The object model assumes that every object is provided with a unique *identity*, which serves as a kind of label to distinguish the object from all others. An object's identity is an intrinsic part of the object model and is distinct from any of the data items stored in the object.

Designers do not need to define a special data value to distinguish individual instances of a class. Sometimes, however, an application domain will contain real data items that are unique to individual objects, such as identification numbers of various kinds, and these data items will often be modelled as attributes. In cases where there is no such data item, however, it is not necessary to introduce one simply for the purpose of distinguishing objects.

In object-oriented programming languages, the identity of an object is usually represented by its address in memory. As it is impossible for two objects to be stored at the same location, all objects are guaranteed to have a unique address, and hence the identities of any two objects will be distinct.

Object names

UML allows objects to be given names, which are distinct from the name of the class the object is an instance of. These names are internal to the model and allow an object to be referred to elsewhere in a model. They do not correspond to any data item that is stored in the object; rather, a name should be thought of as providing a convenient alias for an object's identity.

An object's name is distinct from the name of a variable that happens to hold a reference to the object. When one is illustrating objects it is often convenient, as in Figure 2.2, to use as an object's name the name of a variable containing a reference to that object. However, more than one variable can hold references to the same object and a single variable can refer to different objects at different times, so it would be easy for this convention, if widely applied, to lead to confusion.

Encapsulation

Objects are normally understood to *encapsulate* their data. This means that data stored inside an object can only be manipulated by the operations belonging to that object, and consequently that an object's operations cannot directly access the data stored in a different object.

In many object-oriented languages, a form of encapsulation is provided by the access control mechanisms of the language. For example, the fact that the data members of the 'Part' class in Section 2.2 are declared to be private means that they can only be accessed by operations belonging to objects of the same class. Notice that this class-based form of encapsulation is weaker than the object-based form, which allows no object to have access to the data of any other object, not even that belonging to objects of the same class.

2.4 AVOIDING DATA REPLICATION

Although it is attractively straightforward and simple, it is unlikely that the approach to modelling parts adopted in Section 2.2 would be satisfactory in a real system. Its major disadvantage is that the data describing parts of a given type is *replicated*: it is held in part objects and, if there are two or more parts of the same type, the data will be repeated in each relevant object. There are at least three significant problems with this.

First, it involves a high degree of redundancy. There may be thousands of parts of a particular type recorded by the system, all sharing the same reference number, description and cost. If this data was stored for each individual part, a significant amount of storage would be used up unnecessarily.

Second, the replication of the cost data in particular can be expected to lead to maintenance problems. If the cost of a part changed, the cost attribute would need to be updated in every affected object. As well as being inefficient, it is difficult to ensure in such cases that every relevant object has been updated and that no objects representing parts of a different kind have been updated by mistake.

Third, the catalogue information about parts needs to be stored permanently. In some situations, however, no part objects of a particular type may exist. For example, this might be the case if none had yet been manufactured. In this case, there would be nowhere to store the catalogue information. It is unlikely to be acceptable, however, that catalogue information can only be stored when parts exist to associate it with.

A better approach to designing this application would be to store the shared information that describes parts of a given type in a separate object. These 'descriptor' objects do not represent individual parts. Rather, they represent the information associated with a *catalogue entry* that describes a type of part. Figure 2.3 illustrates the situation informally.

This new design requires that, for each distinct type of part known to the system, there should be a single catalogue entry object which holds the name, number and cost of parts of that type. Part objects no longer hold any data. To find out about a part it is necessary to refer to the catalogue entry object that describes it.

This approach solves the problems listed above. Data is stored in one place only, so there is no redundancy. It is straightforward to modify the data for a given type of part: if the cost of a type of part changed, only one attribute would have to be updated, namely the cost attribute in the corresponding catalogue entry object. Finally, there is no reason why a catalogue entry object cannot exist even if no part objects are associated with it, thus addressing the problem of how part information can be stored prior to the creation of any parts.

2.5 LINKS

The design of the stock control program now includes objects of two distinct classes. Catalogue entry objects hold the information that applies to all parts of a given type, whereas each part object represents a single physical part.

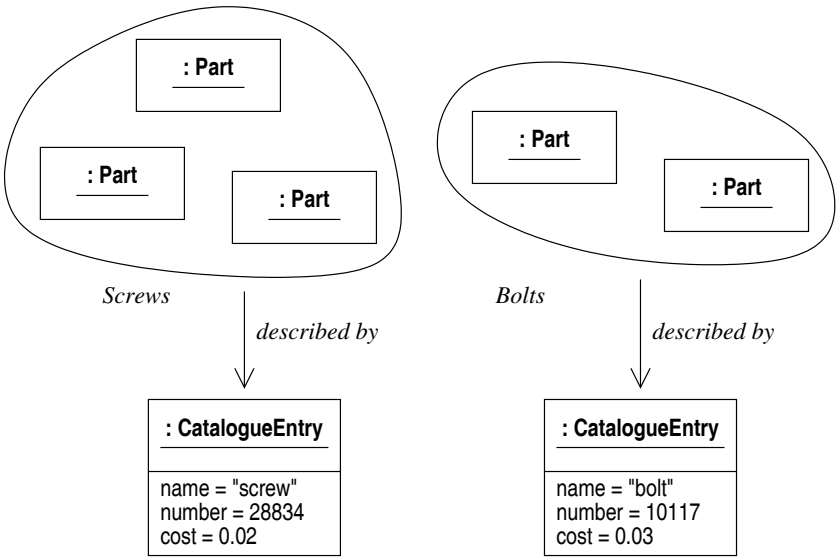

Figure 2.3 Parts described by catalogue entries

There is a significant relationship between these classes, however: in order to obtain a full description of a part, it is necessary to look not only at the part object but also at the related catalogue entry object that describes it.

In practical terms, this means that the system must record, for each part object, which catalogue entry object describes it. A common way of implementing such a relationship is for each part object to contain a reference to the relevant catalogue entry, as shown below. The catalogue entry class now contains the data describing parts, and parts are initialized with, and store a reference to, a catalogue entry.

```
public class CatalogueEntry
{
  private String name ;
  private long   number ;
  private double cost ;

  public CatalogueEntry(String nm, long num, double cst) {
    name   = nm ;
    number = num ;
    cost   = cst ;
  }

  public String getName()   { return name ; }
  public long   getNumber() { return number ; }
  public double getCost()   { return cost ; }
}
```

```
public class Part
{
  private CatalogueEntry entry ;

  public Part(CatalogueEntry e) {
    entry = e ;
  }
}
```

When a part is created, a reference to the appropriate catalogue entry object must be provided. The rationale for this is that it is meaningless to create a part of an unknown or unspecified type. The following lines of code show how part objects are created using these classes. First, a catalogue entry object must be created, but then it can be used to initialize as many part objects as required.

```
CatalogueEntry screw
  = new CatalogueEntry("screw", 28834, 0.02) ;
Part s1 = new Part(screw) ;
Part s2 = new Part(screw) ;
```

As explained in Section 2.2, the fields of a class are often modelled in UML as attributes of the class. However, if a field contains a reference to another object, such as the `entry` field in the `Part` class above, this approach is not appropriate. Attributes define data that is held inside objects, but catalogue entry objects are not held within parts. Rather, they are separate objects which can exist independently of any part objects, and can be referenced from many parts at the same time.

The fact that one object holds a reference to another is shown in UML by drawing a *link* between the two objects. A link is shown as an arrow pointing from the object holding the reference to the object it refers to and it can be labelled at the arrowhead with the name of the field that holds the reference. The objects created by the lines of code above can therefore be modelled in UML as shown in Figure 2.4.

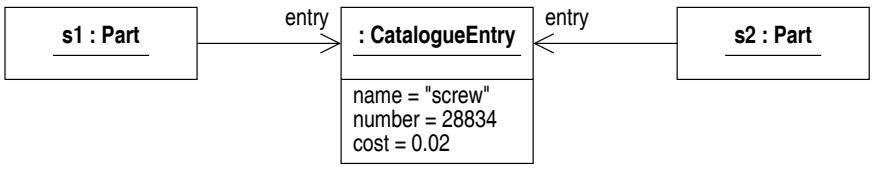

Figure 2.4 Links between objects

The arrowhead on the link indicates that it can only traversed, or *navigated*, in one direction. This means that a part object knows the location of the catalogue entry object it is linked to, and so has access to its public interface. This does not imply that the catalogue entry has any access to, or even any knowledge of, the part objects that reference it. Access in the other direction could only be provided by storing references to parts in catalogue entries, and thereby making the link navigable in both directions.

Object diagrams

An *object diagram* is a diagram that shows objects and links between them. Figure 2.4 is a very simple example of an object diagram. Object diagrams are a way of presenting in a visual form the structure of the object graph discussed in Section 2.1: they give a 'snapshot' of the structure of the data in a system at a given moment.

In a structure of linked objects, information is recorded in two distinct ways. Some data is held explicitly as attribute values in objects, but other information is held purely structurally, by means of links. For example, the fact that a part is of a given type is represented by the link between the part object and the appropriate catalogue entry: there is no data item in the system which explicitly records the type of a part.

2.6 ASSOCIATIONS

Just as the shared structure of a set of similar objects is defined by means of a class, the common properties of the links between those objects can be defined by means of a relationship between the corresponding classes. In UML, a data relationship between classes is called an *association*. Links are therefore instances of associations, in the same way that objects are instances of classes.

In the stock control example, then, the links in Figure 2.4 must be instances of an association between the part and catalogue entry classes. Associations are shown in UML by means of a line connecting the related classes; the association corresponding to the links above is shown in Figure 2.5. Notice that, for the sake of clarity, the diagram does not show all the information known about the catalogue entry class.

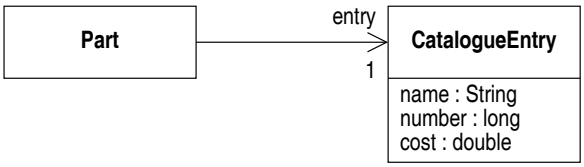

Figure 2.5 An association between two classes

This association models in UML the definition of the `entry` field in the `Part` class. As this field holds a reference, the association is shown as being navigable in only one direction. Like the corresponding links, it is labelled at one end with the name of the field. A label placed at the end of an association in this way is known as a *role name*: its positioning reflects the fact that the name 'entry' is used inside the part class to refer to the linked catalogue entry objects.

The association end is also labelled with a *multiplicity constraint*, in this case the figure '1'. A multiplicity constraint states how many instances a given object can be linked to at any one time. In this case, the constraint is that every part object must be linked to exactly one catalogue entry object. The diagram specifies nothing about how many parts can be linked to a catalogue entry object, however.

Multiplicity constraints give valuable information about a model, but it is important also to be aware of what they do not say. For example, a reasonable property to demand of the stock control system is that a part should always be linked to the same catalogue entry object, as the physical parts being modelled cannot change from one type to another. The multiplicity constraint shown does not enforce this, however: the constraint that there should only be one linked catalogue entry at any given time does not imply that it will be the same one at every time.

It is worth noticing that the Java `Part` class given above does not in fact implement the multiplicity shown in Figure 2.5. As the value `null` is a legal value for a reference field in Java, it would be possible for a part object not to be linked to any catalogue entry, if `null` was provided as an argument to the part constructor. This contradicts the multiplicity constraint that a part should always be linked to exactly one catalogue entry object. A more robust implementation of the `Part` class might check for this at run-time, and perhaps throw an exception if any attempt was made to initialize a part with a null reference.

Class diagrams

Just as object diagrams show collections of objects and links, *class diagrams* contain classes and associations. Figure 2.5 is therefore a simple example of a class diagram.

Whereas object diagrams show particular states of a system's object graph, class diagrams specify in a more general way the properties that any legal state of the system must satisfy. For example, Figure 2.5 states that at any given time the object graph of the stock control system can contain instances of the classes 'Part' and 'CatalogueEntry', and that each part object must be linked to exactly one catalogue entry. If the program gets into a state where a link connects two catalogue entries, say, or a part is not linked to a catalogue entry, an error has occurred and the program is in an illegal state. By a natural terminological extension, object diagrams are referred to as being instances of class diagrams if they satisfy the various constraints defined on the class diagram.

2.7 MESSAGE PASSING

The examples earlier in this chapter have illustrated how data in an object-oriented program is distributed across the objects in the system. Some data is held explicitly as the values of attributes, but the links between objects also carry information, depicting the relationships that hold between objects.

This distribution of information means that typically a number of objects need to interact in order to accomplish any significant functionality. Suppose, for example, that we want to add a method to the `Part` class enabling us to retrieve the cost of an individual part. The data value representing the cost of the part is not held in the part object, however, but in the catalogue entry object that it references. This means that the new method must call the `getCost` method in the catalogue entry class in order to retrieve the data, as shown in the following implementation.

```
public class Part
{
  public double cost() {
    return entry.getCost() ;
  }

  private CatalogueEntry entry ;
}
```

Now if a client holds a reference to a part and needs to find out its cost, it can simply call the `cost` method as follows.

```
Part s1 = new Part(screw) ;
double s1cost = s1.cost() ;
```

UML represents method calls as *messages* that are sent from one object to another. When one object calls another's method, this can be viewed as a request for the called object to perform some processing, and this request is modelled as a message. Figure 2.6 shows the message corresponding to the call `s1.cost()` in the code above.

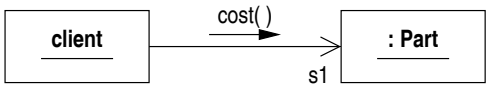

Figure 2.6 Sending a message

The client object in Figure 2.6 has an object name but no class name. The 'cost' message could be sent to a part by objects of many different classes, and the class of the sender of the message is irrelevant to understanding the message and the part object's response to it. It is therefore convenient to omit the class of the client from object diagrams like Figure 2.6, which illustrate particular interactions.

The client code holds a reference to the part object, in the variable `s1`, and this is shown in Figure 2.6 as a link, as before. The reference also enables the client to call the methods of the linked object, however, and this means that in UML links between objects also represent communication channels for messages. Messages are shown on object diagrams as labelled arrows adjacent to links. In Figure 2.6 a client object is shown sending a message to a part object asking it for its cost. Messages themselves are written using a familiar 'function call' notation.

When an object receives a message, it will normally respond in some way. In Figure 2.6 the expected response is that the part object will return its cost to the client object. However, in order to retrieve the cost, the part object must call the `getCost` method in the linked catalogue entry object. This can be represented by a second message, as shown in figure Figure 2.7.

Figure 2.7 also illustrates UML's notation for showing return values from messages. The value returned is written before the message name and separated from it by the assignment symbol ':='. As Figure 2.6 shows, this notation can simply be omitted when the return value is not being shown, or when no value is returned.

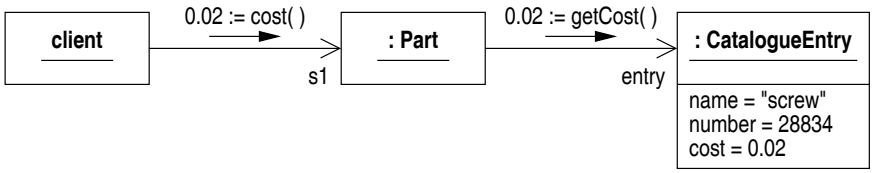

Figure 2.7 Finding the cost of a part

The semantics of the messages shown above are those of normal procedural function calls. When an object sends a message to another object, the flow of control in the program passes from the sender to the object receiving the message. The object that sent the message waits until control returns before continuing with its own processing.

2.8 POLYMORPHISM

As well as maintaining details of individual parts, the stock control program must be capable of recording how they are put together into assemblies. A simple assembly, containing a strut and two screws, is shown in Figure 2.8. Notice that irrelevant attributes of the catalogue entry class have been left out of this diagram.

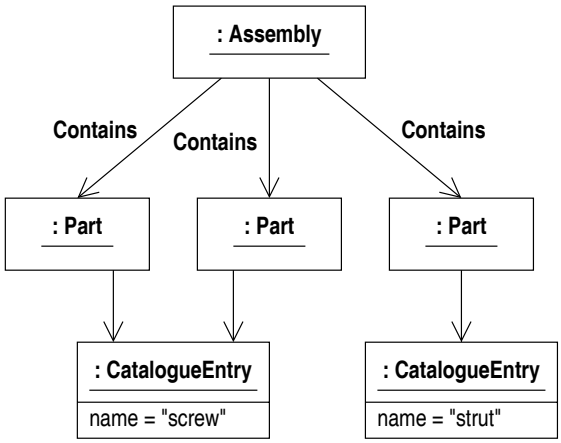

Figure 2.8 A simple assembly

In Figure 2.8 the information about which parts are contained in the assembly is represented by the links connecting the assembly object to the part objects. These links are labelled not with role names but with an *association name*. Association names describe the relationship that holds between the linked objects. The name is often chosen, as here, so that a sentence describing the relationship can be constructed out of the association name and the names of the linked classes. In this case, a suitable sentence might be that 'an assembly contains parts'.

An implementation of the assembly class must provide a way of holding references to an unspecified number of parts. A simple way of supporting this is for the class to contain a data structure that can hold references to all the parts in the assembly, as shown below.

```
public class Assembly
{
  private Vector parts = new Vector() ;

  public void add(Part p) {
    parts.addElement(p) ;
  }
}
```

The association that the links in Figure 2.8 are instances of is shown in Figure 2.9. Like the links, it is labelled with an association name, written in the middle of the association rather than at the end. The symbol '*' at the association end is a multiplicity annotation meaning 'zero or more'. On this diagram, it specifies that an assembly can be linked to, or can contain, zero or more parts.

The diagram in Figure 2.9 also documents the implementation of the association in the code above, by labelling the association end with the role name 'parts', corresponding to the field used to store references in the `Assembly` class. In general, associations and links can be labelled with any combination of name and role name required to make the meaning of the diagram clear.

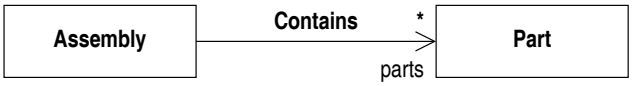

Figure 2.9 An association between assemblies and parts

It is not enough, however, for an assembly to be modelled simply as a collection of parts. Assemblies can have a hierarchical structure, where parts may be assembled into subassemblies, and subassemblies put together with other subassemblies and parts to make higher-level assemblies, to any required degree of complexity. A simple example of this is shown in Figure 2.10, which introduces a subassembly into the structure shown in Figure 2.8.

In order to achieve a hierarchical structure, an assembly must be able to contain both parts and other assemblies. This means that, unlike Figure 2.8, where the links labelled 'Contains' all connected an assembly object to a part object, in Figure 2.10 they can also connect an assembly object to another assembly.

Like many programming languages, UML is strongly typed. Links are instances of associations, and the objects connected by a link must be instances of the classes at the ends of the corresponding association. In Figure 2.8 this requirement is satisfied: every link labelled 'Contains' connects an instance of the assembly class to an instance of the part class, as specified in Figure 2.9.

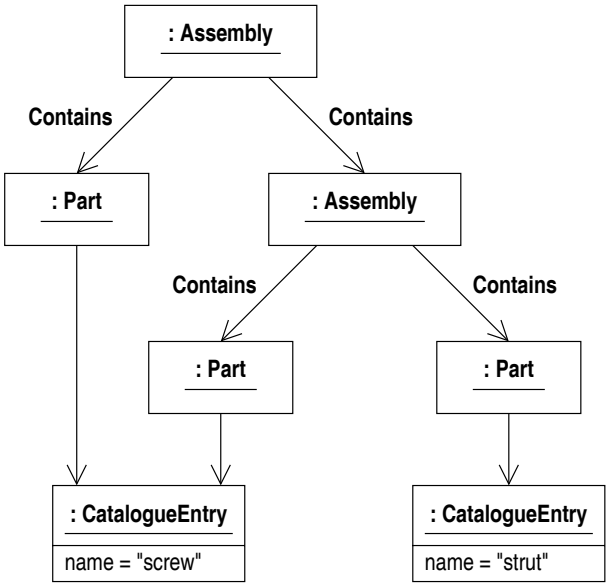

Figure 2.10 A hierarchical assembly

This requirement is violated in Figure 2.10, however, because a 'Contains' link connects the top-level assembly instance not to a part object, but to an instance of the assembly class. If we want to model hierarchical assemblies, the objects at the 'contained' end of these links must not be constrained to belong solely to the 'Part' class, as specified in Figure 2.9, but to either of the 'Part' or 'Assembly' classes. This is an example of *polymorphism*: this word means 'many forms' and suggests that in some situations objects of more than one class need to be connected by links of the same type.

Implementation of polymorphism

UML, being strongly typed, does not allow links to connect objects of arbitrary classes. For polymorphic links like those shown in Figure 2.10, it is necessary to specify the range of classes that can participate in the links. This is normally done by defining a more general class and stating that the specific classes that we wish to link are specializations of the general class.

In the stock control program, we are trying to model situations where assemblies can be made up of *components*, each of which could be either a subassembly or an individual part. We could therefore specify that the 'Contains' links connect assembly objects to component objects, which, by definition, are either part objects or other assembly objects, representing subassemblies.

In object-oriented languages the mechanism of *inheritance* is used to implement polymorphism. A component class could be defined, and the part and assembly classes defined to be subclasses of this class, as shown below.

```
public abstract class Component { ... }

public class Part extends Component { ... }

public class Assembly extends Component
{
  private Vector components = new Vector() ;

  public void add(Component c) {
    components.addElement(c) ;
  }
}
```

Unlike the earlier implementation of the `Assembly` class, which defined a method to add a part to an assembly, the corresponding method here adds a component to the assembly. At run-time, however, the actual objects that are created and added to an assembly will be instances of the part and assembly classes, not of the component class itself. The semantics of inheritance in Java means that references to subclasses can be used wherever a reference to a superclass is specified. In this case, this means that references to both parts and assemblies can be passed as parameters to the add function, as both of these classes are subclasses of the component class, which specifies the type of the function's parameter.

The implementation of the assembly class is potentially even more polymorphic than this, as it uses the Java library class `Vector`, which can hold references to objects of any kind, to store the component references. The restriction to components is enforced by the type of the parameter of the `add` method, which provides the only way that clients can add components to the assembly.

Polymorphism in UML

The implementation of polymorphism in this example arises from the interplay of two different mechanisms. First, the assembly class is defined so that it can hold references to a number of component objects and, second, inheritance is used to define subclasses representing the different type of components that exist. The programming language rules then mean that an assembly can store references to a mixture of the different types of component.

The `extends` relationship of Java is represented in UML by the relationship of *specialization* between classes: if class E is defined by extending class C, then E is said to be a specialization of C. This relationship is also referred to as *generalization* if it is being viewed as the relationship a superclass has to its subclasses: an equivalent description would be to say that C is a generalization of E. Generalization, or specialization, is depicted in UML class diagrams by means of an arrow linking the subclass in the relationship to the superclass. These relationships are distinguished visually from associations by the shape of the arrowhead. Figure 2.11 shows the specialization relationships between the classes in the stock control example.

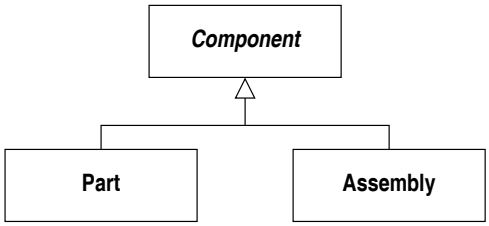

Figure 2.11 Generalization relationships between components

Unlike associations, generalizations do not have 'instances' that appear on object diagrams. Whereas associations describe the ways in which objects can be linked together, generalization describes the situations in which an object of one class can be *substituted* for an object of another. Because of this, the notion of multiplicity is not applicable to generalizations, and generalization relationships are usually not labelled.

Finally, we can redefine the association in Figure 2.9 to take into account the generalization in Figure 2.11. The resulting situation is shown in Figure 2.12. This diagram also documents the implementation of the `Component`, `Part` and `Assembly` classes given above.

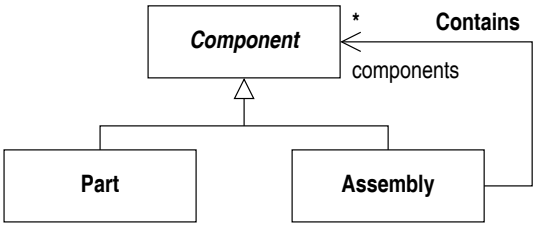

Figure 2.12 A model permitting hierarchical assemblies

Figure 2.12 states that assemblies can contain zero or more components (the association), each of which can be either a part or an assembly (the generalization). In the latter case, we have the situation where one assembly is contained inside another, and therefore this class diagram permits the construction of hierarchical object structures such as the one shown in Figure 2.10.

Abstract classes

Unlike the part and assembly classes, we never expect to create instances of the component class. Parts and assemblies correspond to real objects or constructions in the application domain. The component class, however, is a representation of a concept, namely the fact that parts and assemblies can be considered as specific examples of a more general notion of component. The reason for introducing the component class into the model was not to enable the creation of component objects, but rather to specify that parts and assemblies are in certain circumstances interchangeable.

Classes like ‘Component’, which are introduced primarily to specify relationships between other classes in the model, and not to support the creation of new types of objects, are known as *abstract classes*. As illustrated above, Java allows classes to be declared to be abstract and, in UML, this can be shown by writing the class name in a slanting font, as shown in Figures 2.11 and 2.12.

2.9 DYNAMIC BINDING

If an assembly object is passed a message asking for its cost, it can satisfy this request by asking its components for their cost and then returning the sum of the values it is returned. Components that are themselves assemblies will send similar ‘cost’ messages to their components. Components that are simple parts, however, will send a ‘getCost’ message to the linked catalogue entry object, as shown in Figure 2.7.

Figure 2.13 shows all the messages that would be generated if the assembly object at the top of the hierarchy in Figure 2.10 were sent a ‘cost’ message. Notice how in an object-oriented program a single request can easily give rise to a complex web of interactions between the objects in the system.

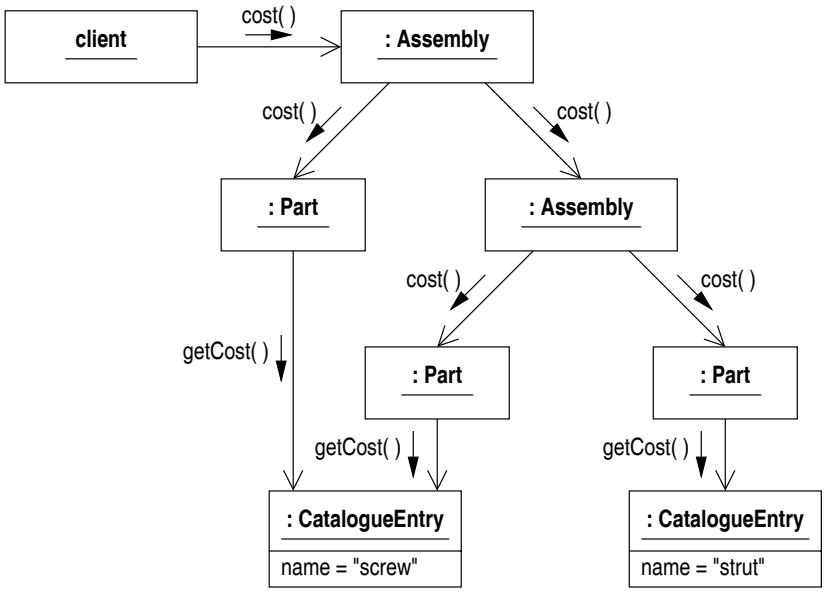

Figure 2.13 Message passing in the hierarchy

In this interaction, assembly objects work out their cost by sending the same message, namely ‘cost’, to all their components. The object sending the message does not know, and in fact need not know, whether a particular component is a part or an assembly. It simply sends the message and relies on the receiving object interpreting it in an appropriate manner.

The actual processing carried out when a 'cost' message is received will depend on whether it has been sent to an assembly or a part object. This behaviour is known as *dynamic*, or *late*, *binding*: essentially, it is the receiver, not the sender, of a message who decides what code is executed as a result of the message being sent. In polymorphic situations, the type of the receiver of a message may not be known until run-time, and hence the code to be executed in response to messages can only be selected at run-time.

In Java, this behaviour is obtained simply by declaring the cost function in the component class and then redefining it in the part and assembly classes to provide the required functionality for each of those classes, as shown in the following extracts from the relevant classes. Other languages have different ways of providing late binding: in C++, for example, the mechanism of virtual functions must be used.

```
public abstract class Component
{
  public abstract double cost () ;
}

public class Part extends Component
{
  private CatalogueEntry entry ;

  public double cost() {
    return entry.getCost() ;
  }
}

public class Assembly extends Component
{
  private Vector components = new Vector() ;

  public double cost() {
    double total = 0.0 ;
    Enumeration enum = components.elements() ;
    while (enum.hasMoreElements()) {
      total += ((Component) enum.nextElement()).cost() ;
    }
    return total ;
  }
}
```

2.10 THE APPLICABILITY OF THE OBJECT MODEL

This chapter has discussed the basic features of the object model, and the close connection between the concepts of the object model and those of object-oriented programming languages has been emphasized. The object model is used at all stages of the software development life-cycle, however, from requirements analysis onwards, and it is important to examine its suitability for these activities.

It is often said that the object-oriented viewpoint is inspired by our everyday way of looking at the world. We do in fact perceive the world as consisting of objects that have various properties, interact with each other and behave in various characteristic ways, and the object model is said to be a reflection of this common sense view. It is sometimes further claimed that, as a result of this, modelling with objects is very easy, and that the objects required in a software system can be found simply by examining the real-world domain being modelled.

It is certainly the case that in some systems there is a fairly direct correspondence between some of the real-world objects being modelled and some of the software objects, but this analogy cannot be taken as a terribly helpful guide to building object-oriented systems. The overriding aims in designing software are to produce systems that meet the users' requirements, are easily maintainable, easy to modify and reuse, and that make efficient use of resources.

In general, however, there is little reason to expect that simply copying the objects perceived in the real world will result in software that has these desirable properties. For example, the straightforward representation of parts as part objects in Section 2.2 was shown in Section 2.4 to lead to significant problems of efficiency and maintainability, and there was even some doubt as to whether it could meet the functional requirements of the system. Another typical example of poor design arising from overemphasizing the properties of real-world objects is given in Section 14.5.

Furthermore, the message-passing mechanism used for communication between objects in the object model does not seem to represent accurately the way that many events happen in the real world. For example, consider a situation where two people, not looking where they are going, bump into each other. It is very counterintuitive to think of this as a situation where one of the people sends a 'bump' message to the other. Rather it seems that an event has taken place in which each person is an equal and unintentional participant. For this kind of reason, some object-oriented analysis methods recommend modelling the real world in terms of objects and events, and only introduce messages relatively late on in the design process.

Clearly there are cases where real-world objects, and in particular agents, can be thought of as sending messages to other objects. The most significant strengths of the object model arise not from its suitability as a technique for modelling the real world, however, but from the fact that programs and software systems that have an object-oriented structure are more likely to possess a number of desirable properties, such as being easy to understand and maintain.

Object-orientation is most helpfully understood as a particular approach to the problem of how to relate data and processing in software systems. All software systems have to handle a given set of data and provide the ability to manipulate and process that data. In traditional procedural systems the data and the functions that process the data are separated. The system's data is thought of as being stored in one place, and the functionality required by an application is provided by a number of operations, which have free access to any part of the data, while remaining essentially separate from it. Each operation has the responsibility of picking out the bits of data that it is interested in from the central repository.

An observation that can immediately be made about this kind of structure is that most operations will only use a small fraction of the total data of the system, and that most pieces of data will only be accessed by a small number of operations. What object-oriented approaches attempt to do is to split up the data repository and integrate pieces of data with the operations that directly manipulate them.

This approach can provide a number of significant technical advantages over more traditional structures. In terms of ease of understanding, however, the benefits of object-oriented design seem to arise not from the fact that the object model is particularly faithful to the structure of the real world, but rather from the fact that operations are localized together with the data that they affect, instead of being part of a large and complex global structure.

2.11 SUMMARY

- Object-oriented modelling languages are based on an abstract *object model*, which views a running system as being a dynamic network of interacting objects. This model also provides an abstract interpretation of the run-time properties of object-oriented programs.
- Objects contain data and a set of operations to manipulate that data. Every object is distinguishable from every other object, irrespective of the data held or operations provided. These properties of objects are known as the *state*, *behaviour* and *identity* of an object.
- A *class* describes a set of objects that share the same structure and properties. These objects are known as the *instances* of the class.
- Objects typically prevent external objects from having access to their data, which is then said to be *encapsulated*.
- *Object diagrams* show a set of objects, at run-time, together with the links between them. Objects can be named, and the values of their attributes can be shown.
- Objects co-operate by sending *messages* to other objects. When an object receives a message it executes one of its operations. Sending the same message to different objects can result in different operations being executed.
- Interactions between objects, in the form of messages, can be shown on object diagrams, together with parameters and return values.
- *Class diagrams* provide an abstract summary of the information shown on a set of object diagrams. They show the same kind of information as is typically found in a system's source code.
- A rule of thumb that is often used in object-oriented modelling is to base a design on the objects found in the real world. The suitability of designs arrived at in this way needs to be evaluated carefully, however.

2.12 EXERCISES

2.1 Draw a complete class diagram describing the final state of the stock control program described in this chapter, including any attributes and operations defined in the code extracts.

2.2 Assume that the following lines of code are executed.

```
CatalogueEntry frame
  = new CatalogueEntry("Frame", 10056, 49.95) ;
CatalogueEntry screw
  = new CatalogueEntry("Screw", 28834, 0.02) ;
CatalogueEntry spoke
  = new CatalogueEntry("Spoke", 47737, 0.95) ;

Part screw1 = new Part(screw) ;
Part screw2 = new Part(screw) ;

Part theSpoke = new Part(spoke) ;
```

(*a*) Draw a diagram showing the objects that have been created, their data members and the links between them.

(*b*) The following code creates an assembly object and adds to it some of the objects created earlier.

```
Assembly a = new Assembly() ;
a.add(screw1) ;
a.add(screw2) ;
a.add(theSpoke) ;
```

Draw a diagram showing the objects involved in the assembly `a` after these lines have executed, and the links between them.

(*c*) Execution of the following line of code can be shown as a `cost()` message being sent to the assembly `a`.

```
a.cost() ;
```

Add onto your diagram the messages that would be sent between objects during execution of this function.

2.3 The object diagrams in Figure Ex2.3 depict impossible states of the stock control program, according to the class diagrams in Figures 2.5 and 2.12. Explain why.

2.4 On a single diagram, illustrate the following using the UML notation for objects, links and messages.

(*a*) An object of class `Window`, with no attributes shown.

(*b*) An object of class `Rectangle` with attributes *length* and *width*. Assume that the rectangle class supports an operation to return the area of a rectangle object.

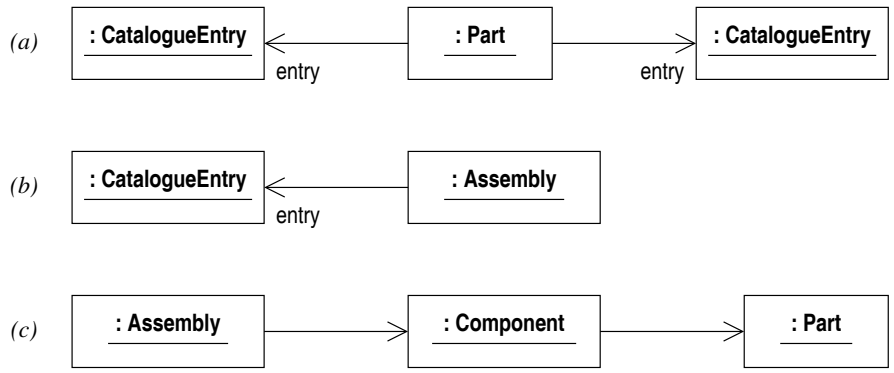

Figure Ex2.3 Illegal states of the stock control program

(*c*) A link between the window and rectangle objects, modelling the fact that the rectangle defines the screen co-ordinates of the window.

(*d*) The window object sending a message to the rectangle asking for its area.

Draw a class diagram showing `Window` and `Rectangle` classes with the properties mentioned in this question.

2.5 Suppose that an environmental monitoring station contains three sensors, namely a thermometer, a rain gauge and a humidity reader. In addition, there is an output device, known as a printer, on which the readings from these three sensors are shown. Readings are taken and transcribed onto the printer every five minutes. This process is known as 'taking a checkpoint'.

(*a*) Draw an object diagram showing a plausible configuration for these objects, and include on the diagram the messages that might be generated in the system every time a checkpoint is taken. Assume that a checkpoint is initiated by a message sent from a timer object to the monitoring station.

(*b*) Does your diagram clearly show the order in which messages are sent? If not, how might this be shown?

(*c*) Draw a class diagram summarizing the structure of the monitoring station.

2.6 A workstation currently has three users logged into it, with account names A, B and C. These users are running four processes, with process IDs 1001, 1002, 1003 and 1004. User A is running processes 1001 and 1002, B is running process 1003 and C is running process 1004.

(*a*) Draw an object diagram showing objects representing the workstation, the users and processes, and links to represent the relationships of a process running on a workstation and a user owning a process.

(*b*) Consider an operation which lists information about the processes that are currently running on a workstation. It can either report on all the current processes or, if invoked with a suitable argument, the processes for a single specified user. Discuss what messages would need to be passed between the objects shown in part (*a*) in order to implement this operation.

2.7 An alternative design for the program discussed in this chapter might do away with the part and catalogue entry classes and instead represent each different type of part by a distinct class. The model could, for example, contain 'screw', 'strut' and 'bolt' classes. The reference number, description and cost of each type of part would be stored as static data members in the relevant class, and individual parts would simply be instances of this class.

(*a*) What difference would this change make to the storage requirements of the program?

(*b*) Define an assembly class for this new design. What assumptions do you have to make to ensure that assemblies can contain parts of different types?

(*c*) Draw a class diagram documenting this new design.

(*d*) It is likely that new types of parts will have to be added to the system as it evolves. Explain how this can be done, first in the original design presented in this chapter and second in the alternative design being considered in this question.

(*e*) In the light of these considerations, which of the two design proposals do you think is preferable, and in which circumstances?

2.8 Suppose that a new requirement is defined for the stock control system, to maintain a count of the number of parts of each type that are in stock and not currently being used in assemblies. Decide where this data should be kept and draw messages on a suitable object diagram to show how the count is decremented every time a part is added to an assembly.

2.9 A 'parts explosion' for an assembly is a report that shows, in some suitable format, a complete list of all the parts in an assembly. Extend the stock management program to support the printing of a parts explosion for an assembly. Illustrate your design by including messages on an object diagram representing a typical assembly, such as the one shown in Figure 2.8.

2.10 In Section 2.5 it was stated that it would be an error to create a part object without linking it to a suitable catalogue entry object. However, the constructor of the part class that was presented in that section did not enforce this constraint, as it did not check whether the catalogue entry reference passed to it was not null. If it was null, a part object would be created that was not linked to any catalogue entry. Extend the constructor defined there to deal with this problem in a sensible way.