Chapter 4
Constructing an Object-Oriented System

4.1 Introduction

This chapter takes you through the design of a simple Object-Oriented system without considering implementation issues or the details of any particular language. Instead, this chapter illustrates how to use Object Orientation concepts to construct a software system. We first describe the application and then consider where to start looking for objects, what the objects should do and how they should do it. We conclude by discussing issues such as class inheritance and answer questions such as "where is the structure of the program?".

4.2 The Application: Windscreen Wipe Simulation

This system aims to provide a diagnosis tutor for the equipment illustrated in Fig. 4.1. Rather than use the wash–wipe system from a real car, students on a car mechanics diagnosis course use this software simulation. The software system mimics the actual system, so the behaviour of the pump depends on information provided by the relay and the water bottle.

The operation of the wash–wipe system is controlled by a switch which can be in one of five positions: off, intermittent, slow, fast and wash. Each of these settings places the system into a different state:

J. Hunt, *A Beginner's Guide to Scala, Object Orientation and Functional Programming*, https://doi.org/10.1007/978-3-319-75771-1_4

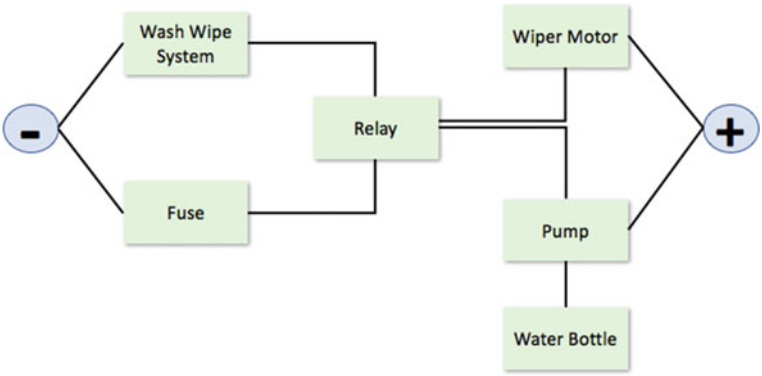

Fig. 4.1 Windscreen wash–wipe system

Switch setting	System state
Off	The system is inactive
Intermittent	The blades wipe the windscreen every few seconds
Slow	The wiper blades wipe the windscreen continuously
Fast	The wiper blades wipe the windscreen continuously and quickly
Wash	The pump draws water from the water bottle and sprays it onto the windscreen

For the pump and the wiper motor to work correctly, the relay must function correctly. In turn, the relay must be supplied with an electrical circuit. This electrical circuit is negatively fused, and thus the fuse must be intact for the circuit to be made. Cars are negatively switched as this reduces the chances of short circuits leading to unintentional switching of circuits.

4.3 Where Do We Start?

This is often a very difficult point for those new to Object-Oriented systems. That is, they have read the basics and understand simple diagrams, but do not know where to start. It is the old chestnut, "I understand the example but do not know how to apply the concepts myself". This is not unusual and, in the case of Object Orientation, is probably normal.

The answer to the question "where do I start?" may at first seem somewhat obscure; you should start with the data. Remember that objects are things that exchange messages with each other. The things possess the data that is held by the system and the messages request actions that relate to the data. Thus, an Object-Oriented system is fundamentally concerned with data items.

Before we go on to consider the Object-Oriented view of the system, let us stop and think for a while. Ask yourself "where would I start if I was going to develop

such a system in C or Pascal or even Ada?" In most cases, the answer is "with some form of functional decomposition". That is, you might think about the main functions of the system and break them down into subfunctions and so on. As a natural part of this exercise, you would identify the data required to support the desired functionality. Notice that the emphasis would be on the system functionality.

Let us take this further and consider the functions we might identify for the example presented above:

Function	Description
Wash	Pump water from the water bottle to the windscreen
Wipe	Move the windscreen wipers across the windscreen

We would then identify important system variables and subfunctions to support the above functions.

Now let us go back to the Object-Oriented view of the world. In this view, we place a great deal more emphasis on the data items involved and consider the operations associated with them (effectively, the reverse of the functional decomposition view). This means that we start by attempting to identify the primary data items in the system; next, we look to see what operations are applied to, or performed on, the data items; finally, we group the data items and operations together to form objects. In identifying the operations, we may well have to consider additional data items, which may be separate objects or attributes of the current object. Identifying them is mostly a matter of skill and experience.

The Object-Oriented design approach considers the operations far less important than the data and their relationships. In the next section we examine the objects that might exist in our simulation system.

4.4 Identifying the Objects

We look at the system as a whole and ask what indicates the state of the system. We might say that the position of the switch or the status of the pump is significant. This results in the data items shown in Table 4.1.

Table 4.1 Data items and their associated state information

Data item	States
Switch setting	Is the switch set to off, intermittent, wipe, fast wipe or wash?
Wiper motor	Is the motor working or not?
Pump state	Is the pump working or not?
Fuse condition	Has the fuse blown or not?
Water bottle level	The current water level
Relay status	Is current flowing or not?

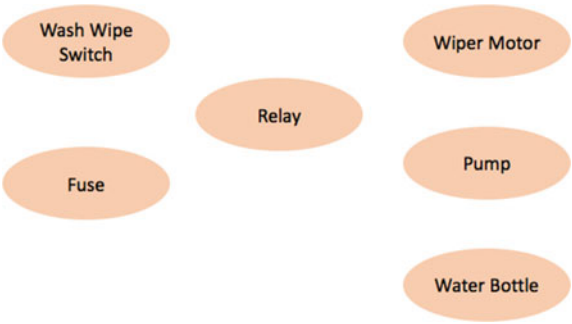

Fig. 4.2 Objects in simulation system

The identification of the data items is considered in greater detail later. At this point, merely notice that we have not yet mentioned the functionality of the system or how it might fit together, we have only mentioned the significant items. As this is such a simple system, we can assume that each of these elements is an object and illustrate it in a simple object diagram (Fig. 4.2).

Notice that I have named each object after the element associated with the data item (e.g. the element associated with the fuse condition is the fuse itself) and that the actual data (e.g. the condition of the fuse) is an instance variable of the object. This is a very common way of naming objects and their instance variables. We now have the basic objects required for our application.

4.5 Identifying the Services or Methods

At the moment, we have a set of objects each of which can hold some data. For example, the water bottle can hold an integer indicating the current water level. Although Object-Oriented systems are structured around the data, we still need some procedural content to change the state of an object or to make the system achieve some goal. Therefore, we also need to consider the operations a user of each object might require. Notice that the emphasis here is on the **user of the object** and what they **require of the object**, rather than what operations are performed on the data.

Let us start with the switch object. The switch state can take a number of values. As we do not want other objects to have direct access to this variable, we must identify the services that the switch should offer. As a user of a switch we want to be able to move it between its various settings. As these settings are essentially an enumerated type, we can have the concept of incrementing or decrementing the switch position. A switch must therefore provide a *moveUp* and a *moveDown* interface. Exactly how this is done depends on the programming language; for now, we concentrate on specifying the required facilities.

If we examine each object in our system and identify the required services, we may end up with the Table 4.2.

Table 4.2 Object services

Object	Service	Description
Switch	moveUp	Increment switch value
	moveDown	Decrement switch value
	state?	Return a value indicating the current switch state
Fuse	working?	Indicate if the fuse has blown or not
Wiper motor	working?	Indicate whether the wipers are working or not
Relay	working?	Indicate whether the relay is active or not
Pump	working?	Indicate whether the pump is active or not
Water bottle	fill	Fill the water bottle with water
	extract	Remove some water from the water bottle
	empty	Empty the water bottle

We generated this table by examining each of the objects in isolation to identify the services that might reasonably be required. We may well identify further services when we attempt to put it all together.

Each of these services should relate to a method within the object. For example, the `moveUp` and `moveDown` services should relate to methods that change the `state` instance variable within the object. Using a generic pseudo-code, the `moveUp` method, within the `switch` object, might contain the following code:

```
define method moveUp()
  if state == "off" then
     state = "wash"
   elseif state == "wash" then
     state = "wipe"
   endif
end define method
```

This method changes the value of the `state` variable in `switch`. The new value of the instance variable depends on its previous value. You can define `moveDown` in a similar manner. Notice that the reference to the instance variable illustrates that it is global to the object. The `moveUp` method requires no parameters. In Object-Oriented systems, it is common for few parameters to be passed between methods (particularly of the same object), as it is the object that holds the data.

4.6 Refining the Objects

If we look back to Table 4.2, we can see that fuse, wiper motor, relay and pump all possess a service called `working?` This is a hint that these objects may have something in common. Each of them presents the same interface to the outside world. If we then consider their attributes, they all possess a common instance variable. At this point, it is too early to say whether fuse, wiper motor, relay and pump are all instances of the same class of object (e.g. a Component class) or whether they are all instances of classes which inherit from some common superclass (see Fig. 4.3). However, this is something we must bear in mind later.

4.7 Bringing It All Together

So far we have identified the primary objects in our system and the basic set of services they should present. These services were based solely on the data the objects hold. We must now consider how to make our system function. To do this, we need to consider how it might be used. The system is part of a very simple diagnosis tutor; a student uses the system to learn about the effects of various faults on the operation of a real wiper system, without the need for expensive electronics.

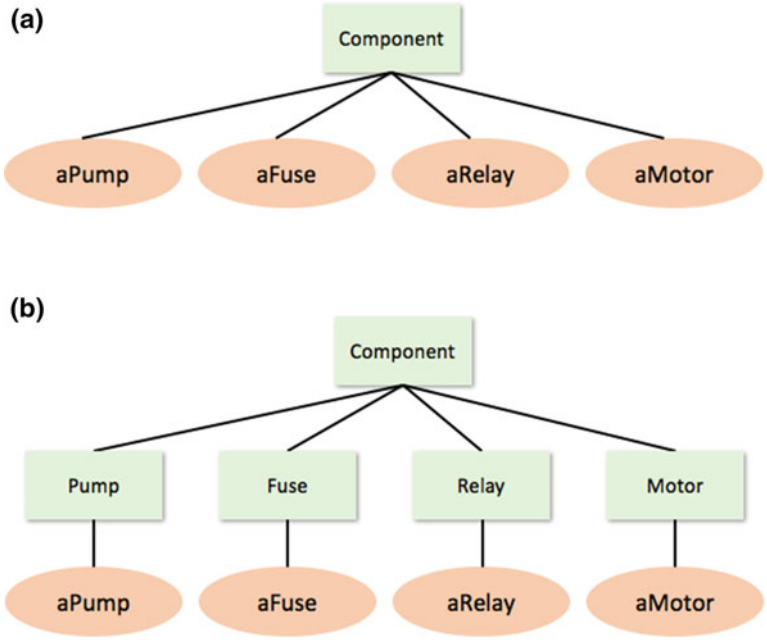

Fig. 4.3 Possible classes for components in the simulation

We therefore wish to allow a user of the system to carry out the following operations:

- change the state of a component device
- ask the motor what its new state is.

The `moveUp` and `moveDown` operations on the switch change the switch's state. Similar operations can be provided for the fuse, the water bottle and the relay. For the fuse and the relay, we might provide a `changeState` interface using the following algorithm:

```
define method changeState()
   if state == "working" then
      state = "notWorking"
   else
       state = "working"
   endif
end define method
```

Discovering the state of the motor is more complicated. We have encountered a situation where one object's state (the value of its instance variable) is dependent on information provided by other objects. If we write down procedurally how the value of other objects affects the status of the pump, we might get the following pseudo-code:

```
if fuse is working then
  if switch is not off then
     if relay is working then
          pump status = "working"
      endif
   endif
endif
```

This algorithm says that the pump status depends on the relay status, the switch setting and the fuse status. This is the sort of algorithm you might expect to find in a `main()` program. It links the subfunctions together and processes the data.

In an Object-Oriented language (such as Scala), we do not have a main program in the same way that a C program has. Instead the `main()` method in Scala is an initiating point for an Object-Oriented system (in Scala this is simplified further by the use of the App trait). As it is part of an object, the main method can trigger the creation of instances, but it is not itself part of those instances. This can be confusing at first; however, if you think of the `main()` method in Scala as initiating a program, that is the starting point for a program, then you are fairly close.

In an Object-Oriented system, well-mannered objects pass messages to one another. How then do we achieve the same effect as the above algorithm? The answer is that we must get the objects to pass messages requesting the appropriate information. One way to do that is to define a method in the pump object that gets the required information from the other objects and determines the motor's state. However, this requires the pump to have links to all the other objects so that it can send them messages. This is a little contrived and loses the structure of the underlying system. It also loses any modularity in the system. That is, if we want to add new components, then we have to change the pump object, even if the new components only affect the switch. This approach also indicates that the developer is thinking too procedurally and not really in terms of objects.

In an Object-Oriented view of the system, the pump object only needs to know the state of the relay. It should therefore request this information from the relay. In turn, the relay must request information from the switches and the fuse.

Figure 4.4 illustrates the chain of messages initiated by the pump object:

1. pump sends a `working?` message to the relay
2. relay sends a `state?` message to the switch
   the switch replies to the relay
3. relay sends a second `working?` message to the fuse
   the fuse replies to the relay
   the relay replies to the motor
   If the pump is working, then the pump object sends the final message to the water bottle
4. pump sends a message `extract` to the water bottle.

In step four, a parameter is passed with the message because, unlike the previous messages that merely requested state information, this message requests a change in

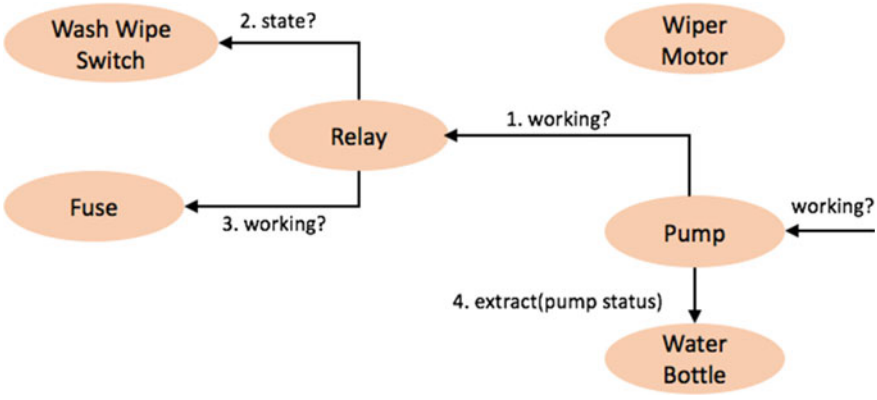

Fig. 4.4 Collaborations between the objects for wash operation

state. The parameter indicates the rate at which the pump draws water from the water bottle.

The water bottle should not record the value of the pump's *status* as it does not own this value. If it needs the motor's status in the future, it should request it from the pump rather than using the (potentially obsolete) value passed to it previously.

In Fig. 4.4, we assumed that the pump provided the service `working?` which allows the process to start. For completeness, the pseudo-code of the `working?` Method for the pump object is:

```
def working?()
  begin
      this.status = relay.working().
       if this.status == "working" then
    water_bottle.extract(this.status)
        endif
    end
end define method
```

This method is a lot simpler than the procedural program presented earlier. At no point do we change the value of any variables that are not part of the pump, although they may have been changed as a result of the messages being sent. Also, it only shows us the part of the story that is directly relevant to the pump. This means that it can be much more difficult to deduce the operation of an Object-Oriented system merely by reading the source code. Some Scala environments (such as the Scala IDE) alleviate this problem, to some extent, through the use of sophisticated browsers.

4.8 Where Is the Structure?

People new to Object Orientation may be confused because they have lost one of the key elements that they use to help them understand and structure a software system: the main program body. This is because the objects and the interactions between them are the cornerstone of the system. In many ways, Fig. 4.4 shows the Object-Oriented equivalent of a main program. This also highlights an important feature of most Object-Oriented approaches: graphical illustrations. Many aspects of object technology, for example, object structure, class inheritance and message chains, are most easily explained graphically.

Let us now consider the structure of our Object-Oriented system. It is dictated by the messages that are sent between objects. That is, an object must possess a reference to another object in order to send it a message. The resulting system structure is illustrated in Fig. 4.5.

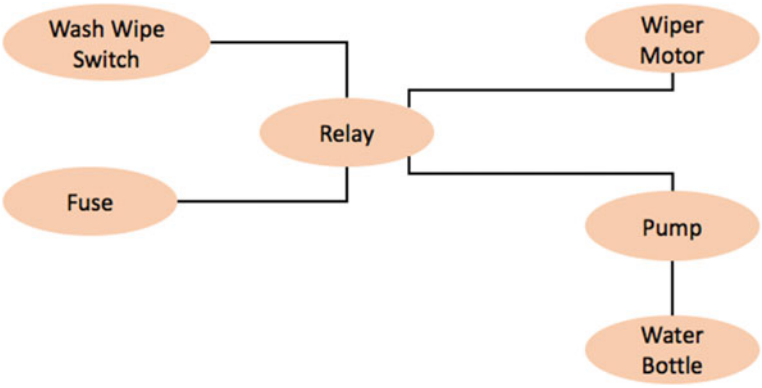

Fig. 4.5 Wash–wipe system structure

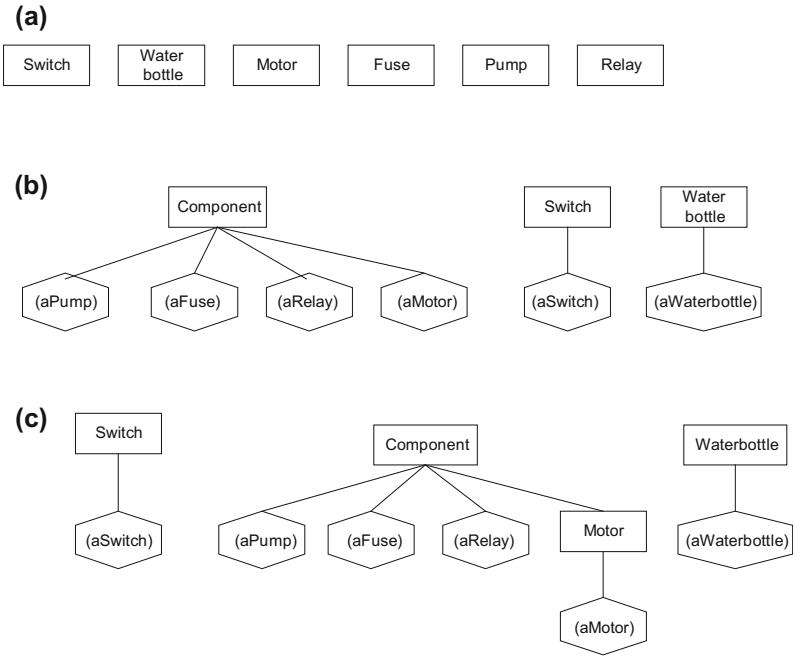

Fig. 4.6 Possible class inheritance relationships

In Scala, this structure is achieved by making instance variables reference the appropriate objects. This is the structure which exists between the instances in the system and does not relate to the classes, which act as templates for the instances.

We now consider the classes that create the instances. We could assume that each object is an instance of an equivalent class (see Fig. 4.6a). However, as has

Table 4.3 Comparison of components

	fuse	relay	motor	pump
Instance variable	state	state	state	state
Services	working?	working?	working?	working?

already been noted, some of the classes bear a very strong resemblance. In particular, the fuse, the relay, the motor and the pump share a number of common features. Table 4.3 compares the features (instance variables and services) of these objects.

From this table, the objects differ only in name. This suggests that they are all instances of a common class such as Component (see Fig. 4.6b). This class would possess an additional instance variable, to simplify object identification.

If they are all instances of a common class, they must all behave in exactly the same way. However, we want the pump to start the analysis process when it receives the message `working?` so it must possess a different definition of `working?` from fuse and relay. In other ways it is very similar to fuse and relay, so they can be instances of a class (say Component) and pump and motor can be instances of classes that inherit from Component (but redefine `working?`). This is illustrated in Fig. 4.6c. The full class diagram is presented in Fig. 4.7.

4.9 Summary

In this chapter, you have seen how a very simple system can be broken down into objects. These objects combine to provide the overall functionality of the system. You have seen how the data to be represented determines the objects used and that the interactions between objects determine the structure of the system. You should also have noted that objects and their classes, methods and instance variables are identified by more of an evolutionary process than in languages that are not Object Oriented.

4.10 Exercises

Take a system with which you are familiar and try to break it down into objects. Carry out a similar set of steps to those described above. Do not worry about how to implement the objects or the classes. Use whatever representation best fits your way of working to describe what the methods do (pseudo-code or a programming language, such as C or Pascal, if you prefer). You can even use a flow chart if you are most comfortable with that. It is very important that you try and do this, as it is a useful exercise in learning to think in terms of objects.

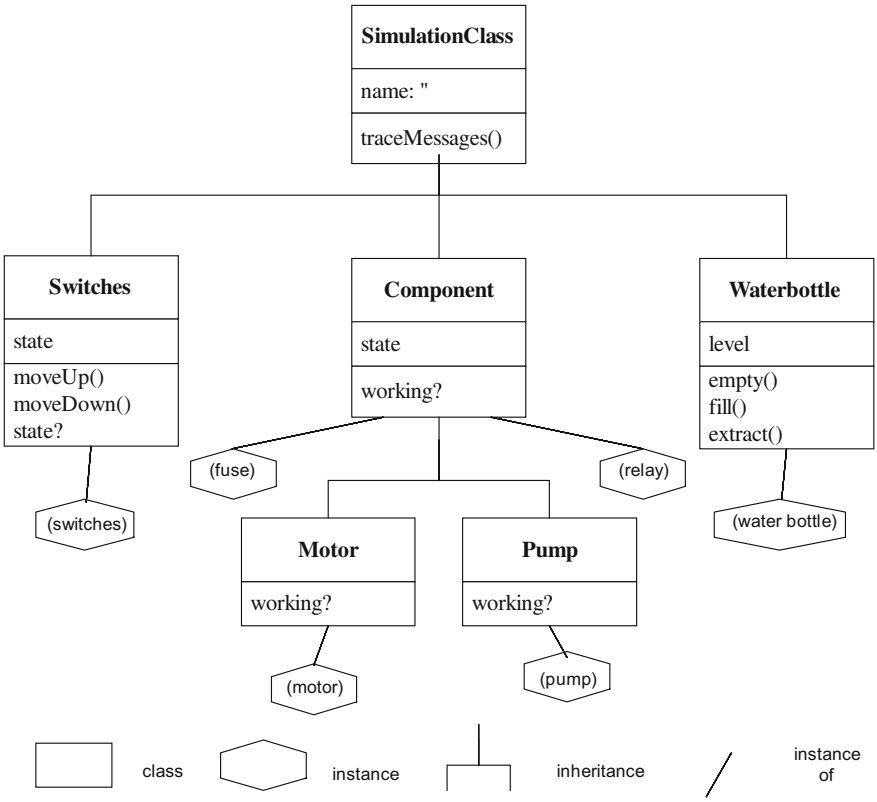

Fig. 4.7 Final class hierarchy and instance diagram

4.11 Further Reading

A good place to start further reading on building Object-Oriented systems is with the first few chapters of Blaha and Rumbaugh (2004). In addition, Wirfs-Brock and McKean (2002) is an excellent, non-language-specific introduction to structuring Object-Oriented systems. It uses a rather simplistic approach, which is ideal for learning about Object-Oriented system design but is not generally applicable. This is not a problem as what you want to do at the moment is to get the background rather than specific techniques. Another good reference for further reading is Larman (2008).

References

Blaha MR, James R (2004) Rumbaugh, Prentice Hall. 8120330161, 2nd edn.

Larman C (2008) Applying UML and patterns: an introduction to object-oriented analysis and design and iterative development. Dorling Kindersley Pvt Ltd.; 3rd edn. 8177589792 (1 Dec 2008)

Wirfs-Brock R, McKean A (2002) Object design: roles, responsibilities and collaborations. Addison-Wesley Object Technologiey Series. 0201379430 (Nov, 2002)