6 Identifying functionality: CRC cards and interaction diagrams

Learning outcomes

The material and exercises in this chapter will enable you to:

- Describe the role of CRC cards in identifying responsibilities and allocating them to classes
- Identify separate operations within a class responsibility
- Explain the purpose of interaction diagrams
- Draw a simple sequence diagram
- Draw a simple collaboration diagram
- Write a process specification to describe the functionality delivered by an operation.

Key words you will find in the glossary:

- algorithm
- class-responsibility-collaboration (CRC) card
- collaboration
- collaboration diagram
- constructor
- decision table
- decision tree
- interaction diagram
- lifeline
- message
- multiobject
- object activation
- operation specification
- package
- reflexive message
- responsibility
- return
- scenario
- sequence diagram
- signature

Introduction

In Chapter 3 we discussed how use case analysis can help identify and document, from the user's perspective, what the system has to do. In Chapters 4 and 5 we discussed how to identify objects and classes and their attributes using noun analysis and how to model relationships between classes. We made a list of the stages in the development of a class diagram and applied the first four stages to the Wheels system. The remaining three stages are: to identify class responsibilities using CRC cards, to separate responsibilities into operations and attributes, and to write process specifications to describe the operations.

In this chapter we discuss how to use the CRC technique to allocate responsibilities to classes and work out the interaction between classes that is required to implement the use case scenarios. We then discuss how to turn these high-level responsibilities into operations on classes and how to describe the functionality of the operations using process specifications. We introduce interaction diagrams and discuss how they are used to document the details of the interactions we identified using the CRC technique. Interaction diagrams also give us a much more precise idea of the associations between classes that are necessary to allow the requisite message passing between objects. In this chapter we also explain the difference between the two types of interaction diagram, sequence and collaboration, and discuss where to use each.

Identifying operations using the CRC card technique

As we saw in Chapter 5, objects can be identified from nouns in a description of the problem domain. In the same way it is possible to identify operations on classes by picking out verbs and verb phrases from the problem description. Phrases that occur in the requirements for the Wheels system (see Figure 5.2), such as 'keep a record of all customers' or 'work out automatically how much it will cost to hire a given bike', tell us that operations will be needed to fulfil these functions. However, analysing verb phrases in this way turns out to be an inefficient method of uncovering operations and allocating them to classes – a more effective and popular approach is to use CRC cards.

CRC (class-responsibility-collaboration) cards are not officially part of the UML, but are regarded as a valuable technique that works extremely well with it. CRC was popularized by a development method called Responsibility Driven Design. The method and the book that describes it (Wirfs-Brock *et al.*, 1990) are both quite old now, but the idea of thinking about a class in terms of the responsibilities it has to fulfil and the technique of CRC cards

are both well regarded and widely used in object-oriented development today. We will look first at the concept of a responsibility and how this leads to identifying operations, and we will then describe how the CRC card technique works in practice.

Class responsibilities and operations

At this stage of constructing a class diagram, our aim is to look at the overall functionality of the system, as identified in the use cases, and divide it up between the classes that we have identified in the class diagram. Each class is regarded as having certain responsibilities to provide services to the user of the system (such as maintaining a customer record) or to another class (such as supplying data for a calculation). Each responsibility identifies something that a class is expected to do; it is an obligation on the class to provide some kind of service.

CRC cards

The aim of the CRC card technique is to divide the overall functionality of the system into responsibilities which are then allocated to the most appropriate classes. Once we know the responsibilities of a class, we can see whether it can fulfil them on its own, or whether it will need to collaborate with other classes to do this.

The actual CRC cards are usually index cards about 10 cm × 15 cm in size, and each card represents one class in the system. The size of the card is important because it restricts the amount that can be written on it. On the front of the card is a high-level description of the class, and the back of the card records the class name, responsibilities and collaborations (if any). An example of a CRC card for the Customer class in the Wheels system is shown in Figure 6.1. We can see from the figure that the Customer class has two responsibilities; it is able to carry out the first of these (Provide customer information) on its own, but in order to carry out the second responsibility (Keep track of hire transactions) it will have to collaborate with the Hire class.

Customer	
Responsibility	*Collaborator*
Provide customer information	
Keep track of hire transactions	Hire

Figure 6.1 CRC card for the Customer class in the Wheels system

This size of card is used rather than a sheet of A4 paper to encourage system developers to restrict the size of each class in terms of the number of its responsibilities. Good object-oriented design depends on having small cohesive classes, each of which has limited, but well-defined functionality. The CRC card also encourages developers to specify responsibilities at a high level rather than write lots of low-level operations. The general rule is that no class should have more than three or four responsibilities.

One of the most effective ways of using CRC cards is in group role-play. Each member takes the role of an object of one of the classes in the system, and the group then enacts the events that take place during a typical scenario. As each responsibility arising from the scenario is identified, it is allocated to the most suitable object. If a responsibility cannot be fulfilled by the existing objects, a new object (and therefore class) will have to be identified. The aim is to minimize the number and complexity of the messages that need to be passed between the objects, and ultimately to produce classes that have a clear purpose and are internally coherent. This method of using CRC cards is popular with both developers and clients as it tends to promote ideas and facilitate discussions. CRC cards do not involve any special notation and so are easily accessible to clients and users of the system.

Figure 6.2 (repeated from Chapter 3) shows a typical scenario from the 'Issue bike' use case. In this figure, where we identify a responsibility of the system, we have added the object (in bold, e.g. **:Bike**) that will carry it out. This illustrates how the overall functionality of the 'Issue bike' use case will be divided up between the classes in the system.

Although it does not feature in this scenario, which models only the user's view of the system, we also know that we need a :Hire object to record details about the hire transaction (see requirement R4 in Figure 5.2: record the details of a hire transaction including the start date, estimated duration, customer and bike). Collectively, the objects that interact to execute a use case are known as a collaboration; the collaboration for the 'Issue bike' use case is shown in Figure 6.3.

Software developers find that the simplicity of CRC cards, their lack of detail, make them an ideal tool for exploring alternative ways of dividing responsibilities between classes. It's easy to scrap one design and start again without agonizing about the amount of work that is being discarded. Another way of exploring alternatives is to use sequence diagrams, which are discussed later in this chapter. In practice, however, sequence diagrams show too much detail and are too slow to draw to be useful for this purpose. They are much more useful for documenting the detail of design decisions once these have been reached using CRCs.

- Stephanie arrives at the shop at 9.00am one Saturday and chooses a mountain bike
- Annie sees that its number is 468
- Annie enters this number into the system
- The system confirms that this is a woman's mountain bike and displays the daily rate (£2) and the deposit (£60) **(:Bike)**
- Stephanie says she wants to hire the bike for a week
- Annie enters this and the system displays the total cost £14 + £60 = £74 **(:Bike)**
- Stephanie agrees this
- Annie enters Stephanie's name, address and telephone number into the system **(:Customer)**
- Stephanie pays the £74
- Annie records this on the system and the system prints out a receipt **(:Payment collaborating with :Customer)**
- Stephanie agrees to bring the bike back by 5.00pm on the following Saturday.

Figure 6.2 *Scenario for the 'Issue bike' use case showing the objects that will carry out the different responsibilities*

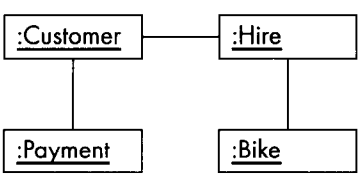

Figure 6.3 *Collaboration of objects required by the 'Issue bike' use case scenario in Figure 6.2*

Applying the CRC method to the Wheels classes in Figure 6.3 yields the set of responsibilities shown in Figure 6.4.

Deriving operations from CRC responsibilities

As we start to move from analysis to design we need more detail about how class responsibilities are going to be carried out. This means that we need to specify the responsibilities in terms of individual operations and attributes; we must ensure that each class has the data and operations needed to fulfil its responsibilities in the way that the user requires. We demonstrate this for the

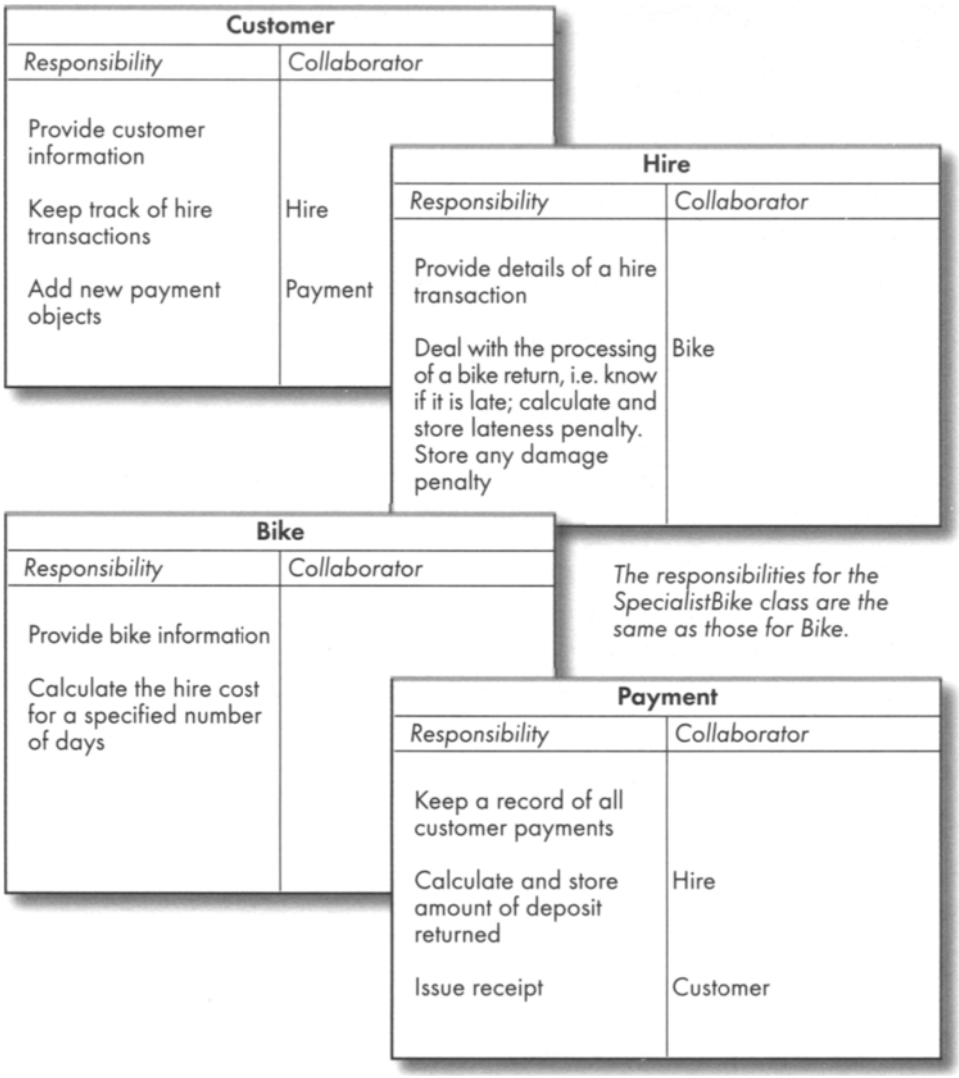

Figure 6.4 *CRC cards for the Wheels system*

'Issue bike' use case, using the use case description to guide us (see Figure 6.5), as it is more general than the scenario.

From the use case description in Figure 6.5, we can see that issuing a bike involves the following tasks.

1 We need to be able to input to the system a bike number (all numbers are stencilled on the bikes) and retrieve the details of this bike to confirm that it is the bike the customer has chosen and to inform the customer of the hire and deposit charges. To do this, we need an operation findBike(), which takes bike number as a parameter, findBike(bike#). We know from the CRC analysis that

Use case: Issue bike
Preconditions: 'Maintain bike list' must have been executed
Actors: Receptionist
Goal: To hire out a bike

Overview:
When a customer comes into the shop they choose a bike to hire. The Receptionist looks up the bike on the system and tells the customer how much it will cost to hire the bike for a specified period. The customer pays, is issued with a receipt, then leaves with the bike.

Cross-reference:
R3, R4, R5, R6, R7, R8, R9, R10

Typical course of events:

Actor action	System response
1 The customer chooses a bike	
2 The Receptionist keys in the bike number	3 Displays the bike details including the daily hire rate and deposit
4 Customer specifies length of hire	
5 Receptionist keys this in	6 Displays total hire cost
7 Customer agrees the price	
8 Receptionist keys in the customer details	9 Displays customer details
10 Customer pays the total cost	
11 Receptionist records amount paid	12 Prints a receipt

Alternative courses:

Steps 8 and 9 — The customer details are already in the system so the Receptionist needs only to key in an identifier and the system will display the customer details.

Steps 7–12 — The customer may not be happy with the price and may terminate the transaction

Figure 6.5 Use case description for 'Issue bike'

providing information about bikes is the responsibility of the Bike class, so findBike() should be an operation on Bike, allowing us to retrieve information from the appropriate bike object.

2 We then want to be able to find out the hire cost for a specific number of days' hire. This again is the responsibility of the Bike class, so we need another operation on Bike, getCharges(no.Days).

3 The next step is to record the customer details. Knowing about customers is the responsibility of the Customer class. We need an operation recordDetails(custID, name, address, tel).

Alternatively, we could do this by using a constructor, which is an operation that creates a new object of a class.

4 The system is required to record what the customer has paid, both in hire charges and as a deposit. The system must also print out a receipt, sometimes for more than one bike. These are both the responsibility of the Payment class. We need an operation on Payment, calcTotalPayment(), which will add up all the hire fees and put them in the Payment attribute totalAmountPaid, and add up all the deposits due and put them all in totalDepositPaid. We also need an operation issueReceipt() that will print out a receipt in the required format. If we want to print the customer's name and address on the receipt, we need to retrieve these from the relevant Customer object.

5 Requirement R4 (see Figure 5.2) states that we must record the details of a hire transaction including the start date and estimated duration. This is not part of the user's view of the system, so is not mentioned in the use case description. However, it should be done at an appropriate point in the proceedings. It is the responsibility of the Hire class to record these details; we need an operation to create a new Hire object and record the details. This could be done by a constructor Hire() with the parameters startDate, no.Days.

By working through the responsibilities identified from the CRC cards, we have identified the operations needed to fulfil the functionality of the 'Issue bike' use case and allocated them to the appropriate classes.

At this stage, the classes and operations are as follows. We have added constructors for all of the classes.

Bike	findBike(bike#)
	getCharges(no.Days)
	Bike()
Customer	recordDetails(custID, name, address, tel)
	Customer()
Payment	calcTotalPayment()
	issueReceipt()
	Payment()
Hire	Hire(startDate, no.Days)

We now need to work through all the other use cases in the same way to identify all the operations required to fulfil the functionality of the system and then add these to the class diagram that we drew in Chapter 5 (see Figure 5.15). The class diagram with all the attributes and operations is shown in Figure 6.6.

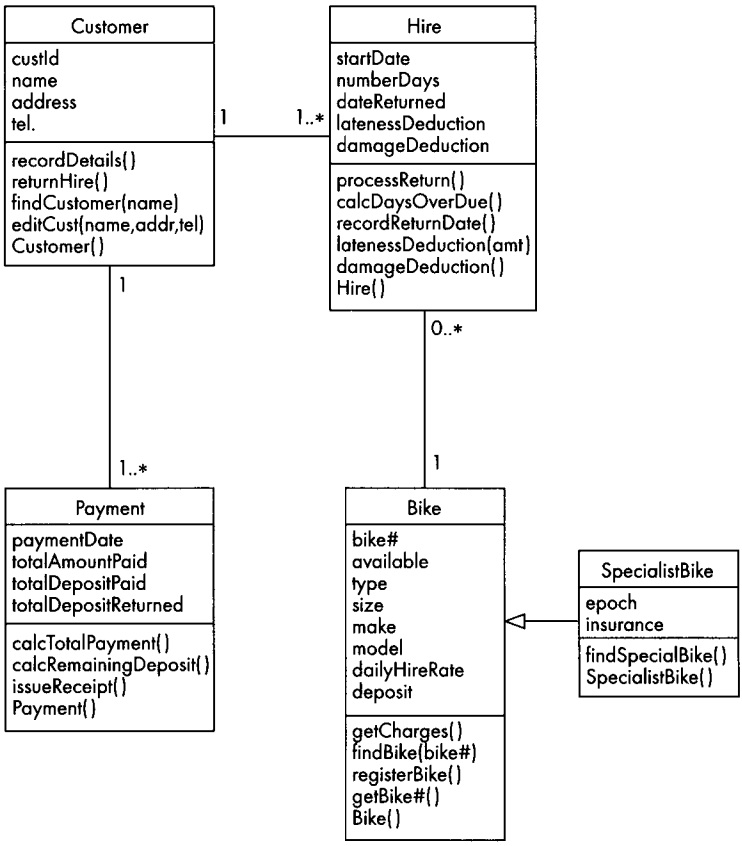

Figure 6.6 Completed class diagram with attributes and operations

Interaction diagrams

We have already mentioned in Chapter 4 that objects collaborate to achieve the functionality required of them by sending messages. Interaction diagrams model the messaging between a collaboration of objects that will take place in the execution of a specific scenario.

By the time we come to do the interaction diagrams we have already done a lot of analysis: we know what the system has to do, we know about the classes and attributes, and we have had our first real ideas about the nature of the relationships between classes – they have to be able to support the collaborations between objects that we specified in the CRC analysis. We have also refined the broad outlines of the class responsibilities into a set of operations and attributes that enable the classes to fulfil their responsibilities.

However, although we now have an idea of which objects are needed to produce the required behaviour and a high-level view of how objects need to collaborate to do this, we have no real idea of the sequence of messages this involves. Use case scenarios describe the functionality of a use case in terms of a sequence of events. We

need to revisit the sequence of events in each scenario, this time looking at them in terms of the messages between objects spawned by each event. This is what interaction diagrams do. There are two types of interaction diagram: sequence and collaboration. Each shows more or less the same information, but with a different emphasis. We shall describe each in turn.

Sequence diagrams

When students who are new to object-oriented technology, especially those who have been trained in procedural methods, first meet object-oriented code, they are staggered by the way the flow of control jumps about on the page. In procedural programming, the sequence of execution proceeds in an orderly fashion from the top of a page of code to the bottom, with only the odd jump off to a procedure. Quite the opposite happens in the execution of an object-oriented program; the sequence of control jumps from object to object in an apparently random manner. This is because, while the code is structured into classes, the sequence of events is dictated by the use case scenarios. Even for those experienced in object technology, it is very hard to follow the overall flow of control in object-oriented code. Programmers, maintainers and developers need a route map to guide them; this is provided by the sequence diagram. The specification of the functionality in a sequence diagram is literally a sequence of messages, but the object-oriented code underlying the sequence of functionality in the diagram jumps about because the code is structured into classes not functions.

Sequence diagrams show clearly and simply the flow of control between objects required to execute a scenario. A scenario outlines the sequence of steps in one instance of a use case from the user's side of the computer screen, a sequence diagram shows how these steps translate into messaging between objects on the computer's side of the screen. We will illustrate this by walking through a simple scenario, Figure 6.7 (repeated from Figure 6.2), translating it into a sequence diagram.

As we saw in Figure 6.3 the objects required by the 'Issue bike' use case are :Bike, :Customer, :Hire and :Payment. The sequence diagram displays this collaboration horizontally across the page as in Figure 6.8.

The order in which the objects appear is not important. The Receptionist icon is the actor involved in the 'Issue bike' use case. As far as interaction diagrams are concerned, actor is treated as a special sort of object. On a sequence diagram produced during analysis she represents the system interface – she inputs information and receives the output from the system. The dashed vertical line below each object symbol is called the object's *lifeline*. It represents the object's life for the duration of the scenario we are enacting. A message is represented as a labelled arrow from one

- Stephanie arrives at the shop at 9.00am one Saturday and chooses a mountain bike
- Annie sees that its number is 468
- Annie enters this number into the system
- The system confirms that this is a woman's mountain bike and displays the daily rate (£2) and the deposit (£60)
- Stephanie says she wants to hire the bike for a week
- Annie enters this and the system displays the total cost £14 + £60 = £74
- Stephanie agrees this
- Annie enters Stephanie's name, address and telephone number into the system
- Stephanie pays the £74
- Annie records this on the system and the system prints out a receipt
- Stephanie agrees to bring the bike back by 5.00pm on the following Saturday.

Figure 6.7 Successful scenario for the use case 'Issue bike'

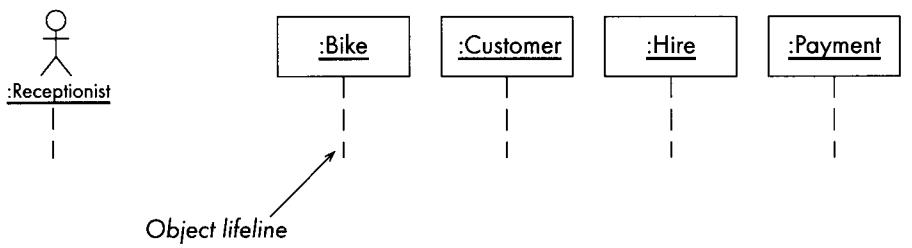

Figure 6.8 Collaboration objects for 'Issue bike' in sequence diagram format

object lifeline (the sender) to another (the recipient). The sequence of messages is read from the top of the page to the bottom, i.e. the time ordering of the messages goes from top to bottom.

The convention for object labelling is the same as for object diagrams (see Chapter 4), i.e. objectName :ClassName where either objectName or :ClassName may be omitted.

As an example, we will now convert the scenario in Figure 6.7 into a sequence diagram. This process is described in the steps below.

1 Stephanie arrives at the shop at 9.00am one Saturday and chooses a mountain bike

2 Annie sees that its number is 468

3 Annie enters this number into the system

4 The system confirms that this is a woman's mountain bike and displays the daily rate (£2) and the deposit (£60)

- Steps 1 and 2 do not require the system to do anything.
- In step 3 Annie, in her role as Receptionist, enters the bike number into the system. We represent this step in a sequence diagram (Figure 6.9) using the findBike() operation we identified from the class responsibilities.

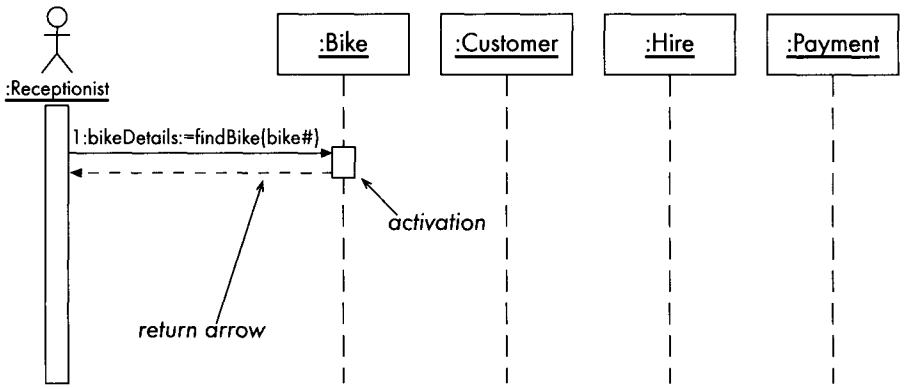

Figure 6.9 Fragment of sequence diagram

- The Receptionist sends the message findBike(bike#) to the relevant bike object, the one whose bike number matches the one she has input.
- For this to work, the target object, :Bike, must understand the message. This means that it must correspond to an operation on the Bike class.
- Sometimes we want to show the value that is returned in response to a message. Returned values are usually shown on the message line. The returned value is assigned to a variable. In this case the value returned by findBike() is assigned to bikeDetails (see step 4).
- The message has a number. On sequence diagrams the numbers are optional as the order is implicit in the sequence in which messages are drawn.
- Return of control can be indicated in UML by a dashed arrow. Return arrows are optional. Notice that the return arrow does not have a number, it is not a new message but models the return of control to the sending object.
- Object *activation* is shown by a thin rectangle on the object's lifeline. An object becomes active as soon as it receives a message. This means that the object is computing, processing is taking

place in the object as the invoked operation executes. The activation continues until the operation finishes processing when control returns to the object that sent the message. If an active object, in turn, sends a message, it remains active while it waits for a response. While it waits for a response it cannot, itself, do any computing. Showing activation is an optional feature of sequence diagrams.

- A lot of detail is omitted on a diagram that is drawn during analysis. It says nothing about how the Receptionist manages to send a message to the :Bike. In fact when we get to the design version of this diagram, we will use an interface object, which will offer options to the user and translate them into messages to objects.
- The analysis diagram also omits any detail about how the matching bike object is found. At this stage we can take it on trust that it is found and leave the detail of how this happens to the designer (see Chapter 9).

The next two steps are:

5	Stephanie says she wants to hire the bike for a week
6	Annie enters this and the system displays the total cost £14 + £60 = £74.

Using the getCharges() operation with the parameter (no.Days) (see Figure 6.10) we can ask :Bike to work out the cost for the required period. The system is organized in such a way that the customer has the opportunity to see the total cost before being committed to the transaction. Only once the customer has agreed to the cost do we record details about the customer and the hire.

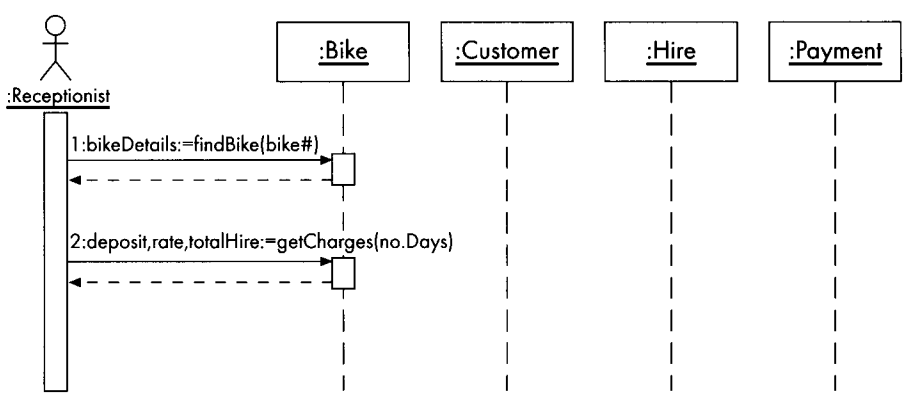

Figure 6.10 Fragment of sequence diagram

The getCharges() operation calculates and returns the deposit, daily hire rate and total amount to be paid (deposit + (no.Days * dailyHireRate)). deposit and dailyHireRate are attributes of :Bike.

The next few steps in the scenario:

7 Stephanie agrees this
8 Annie enters Stephanie's name, address and telephone number into the system.

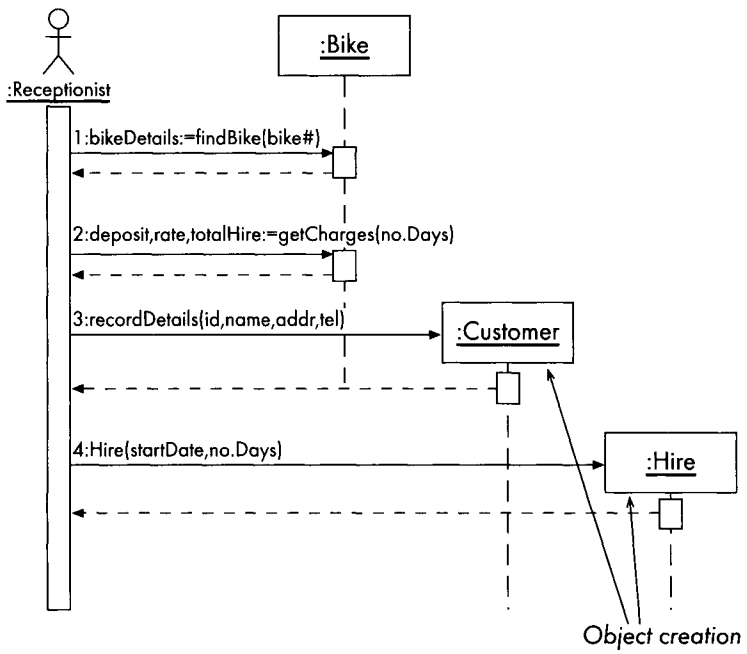

Figure 6.11 *Fragment of sequence diagram with :Customer and :Hire added*

- Once Stephanie has agreed to hire the bike at the rates quoted (which does not affect the system), we can start recording her details.
- Stephanie is a new customer, so the system must create a new Customer object and send it details about Stephanie (see Figure 6.11). We use the recordDetails() operation to do this with the arguments: id, name, addr, tel.
- The UML notation for a new object is to show the creating operation being sent to the object symbol rather than its lifeline.
- Although this is not mentioned in the scenario, we also have to record the details about the hire. This means we need to create a new Hire object and record the start date of the hire and its

duration in days. We can do this using the Hire class constructor, Hire(startDate, no.Days).

- Again, a lot of detail is omitted from the diagram. We are not saying anything about where the parameters for the hire constructor come from. We wouldn't want the Receptionist to have to enter the number of days again, so presumably, some sort of interface or control object will deal with holding on to these details when they were entered the first time.

The last three steps in the scenario are: .

9 Stephanie pays the £74

10 Annie records this on the system and the system prints out a receipt

11 Stephanie agrees to bring the bike back by 5.00pm on the following Saturday

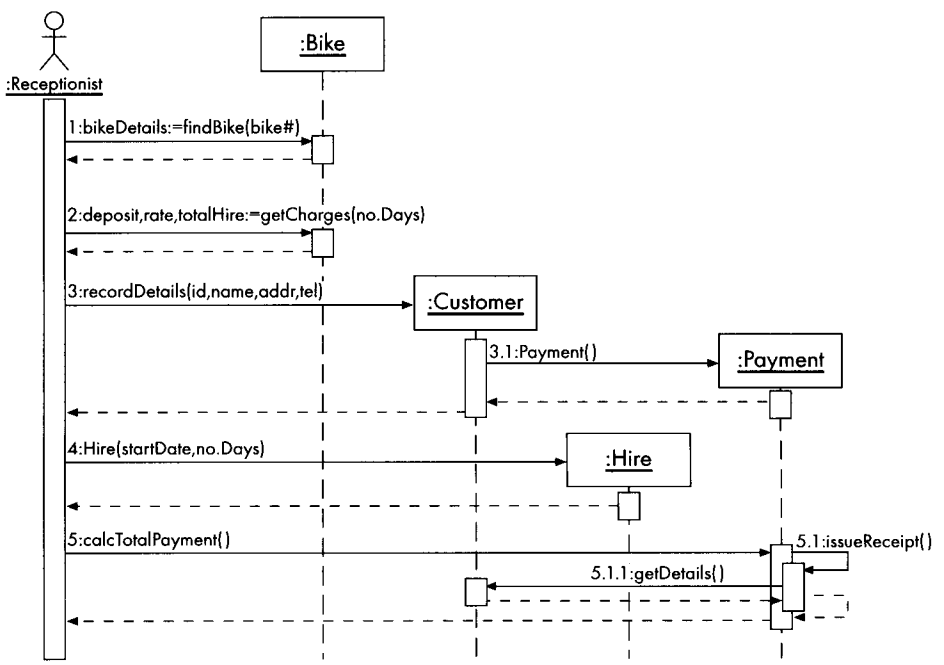

Figure 6.12 Complete sequence diagram for the 'Issue bike' scenario

- The system is required to record what the customer has paid, both in hire charges and as a deposit. A Payment object will calculate how much the customer owes and record these figures. It makes sense to create a Payment object from the Customer object, so that they are permanently linked, see Figure 6.12.

- Notice that the message from :Customer to :Payment is numbered 3.1 rather than 4. The UML numbering style for interaction diagrams emphasizes the nesting of the messages, rather than the

numerical sequence. This is to make it clear which object is calling which.

- calcTotalPayment() will scoop up the total hire fees and total deposits paid and record them in totalAmountPaid and totalDepositPaid. How it does this is left to the designer.
- As it executes, the calcTotalPayment() operation calls issueReceipt() which is another operation on the Payment class; this is known as a *reflexive message*. The Payment object is already active, so this second activation is shown as a new activation box on top of the existing one.
- As the issueReceipt() operation is executing, it sends a message to :Customer to retrieve the customer name and address so that it can print them on the receipt.
- In Figure 6.12 all returns are shown so that you can clearly see the nesting of operations and how control returns eventually to the sending objects. In fact return arrows are usually omitted unless they add meaning to the diagram. Control is always returned to the sending object as soon as the receiving object ceases to be active. The activation boxes tell us how long an object is active, so we can assume the returns. Figure 6.13 shows us the same diagram without the return arrows.
- Notice that :Customer remains active while it sends a message to :Payment. It ceases to be active only when it returns control to the interface.

Collaboration diagrams

Collaboration diagrams show pretty much the same information as sequence diagrams. In fact most CASE tools will automatically generate a collaboration diagram from a sequence diagram or vice versa.

The collaboration diagram version of Figure 6.13 is shown in Figure 6.14.

Although this diagram contains much the same information as the diagram in Figure 6.13, it is obviously different in a number of ways.

- Messages are not shown in time sequence, so numbering becomes essential to indicate the order of the messages
- Links between objects are explicitly modelled, which is not the case with the sequence diagram
- Messages are grouped together on the object links
- Messages are shown as text labels with an arrow that points from the client (the object sending the message) to the server (the object providing the response)

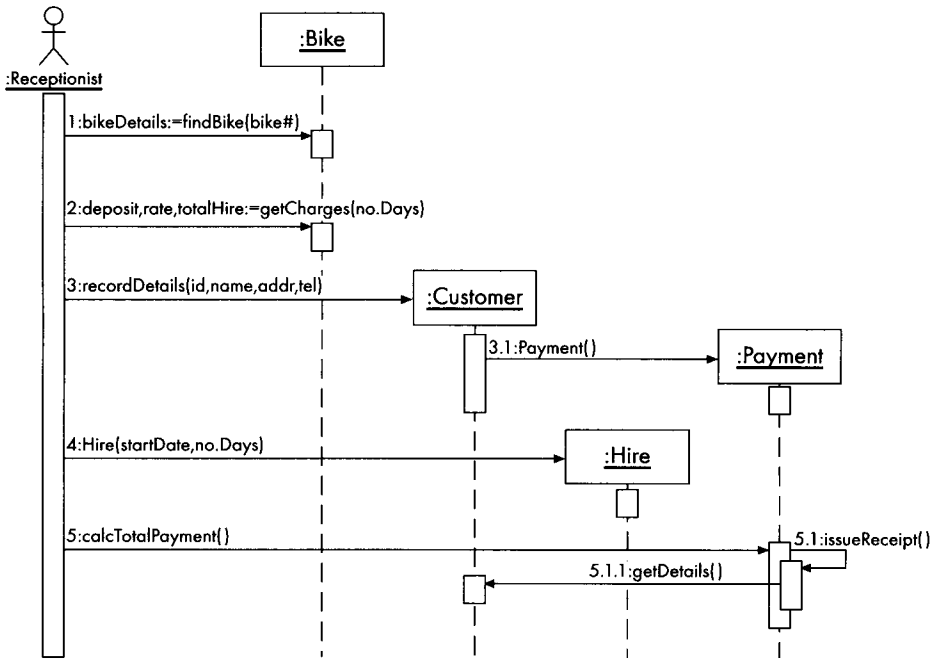

Figure 6.13 Complete sequence diagram for the 'Issue bike' scenario without returns

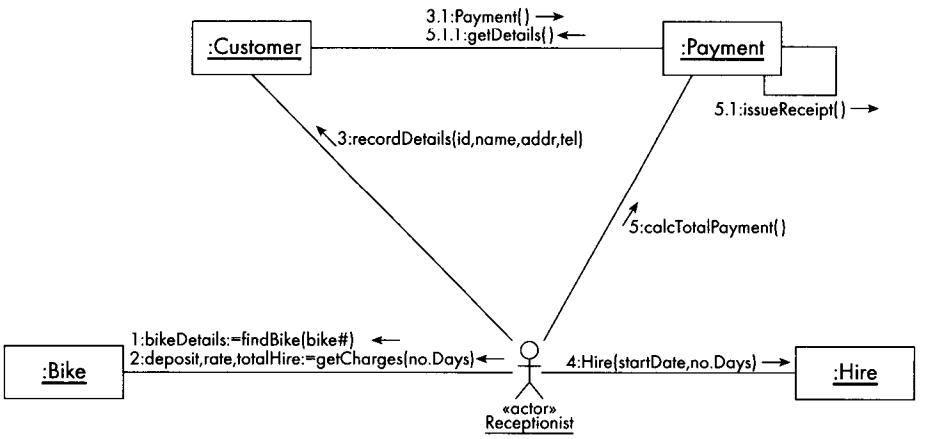

Figure 6.14 Collaboration diagram for the 'Issue bike' scenario

- Unlike sequence diagrams, returns are never modelled in a collaboration diagram
- In Figure 6.14, the message issueReceipt() is a reflexive message, i.e. a message from :Payment to itself. A link is modelled from :Payment to itself and the message goes along the link. On the sequence diagram this is shown as a nested activation.

Using sequence and collaboration diagrams

As sequence and collaboration diagrams are logically equivalent (they display the same information), there is no point in drawing both at any given stage. Both types of diagram convert a textual scenario into a graphical view of the flow of events, and both can be shown at varying levels of detail. If either diagram gets too cluttered with messages we can choose to model only the main flow of messages. Both diagrams can be used to represent the functionality of the system at different levels, for example to illustrate how a use case is realized or to show the workings of a complicated operation.

The main advantage of the sequence diagram is its ability to represent the passage of time graphically. The order of messages is very clear: a sequence diagram reads from top to bottom. It is, of course, possible to figure out the sequence of messages from the numbers on a collaboration diagram, but it is not so intuitively clear. Sequence diagrams can also include return arrows; collaboration diagrams never show return arrows. Another feature that can be added to a sequence diagram is object activation, showing when the object is active. Collaboration diagrams don't have the equivalent of activations.

The special feature of collaboration diagrams is that they include explicit links between objects. A message from one object to another means that there should be an association between the classes to which they belong. In a collaboration diagram this association between classes is represented by an explicit link between the objects of the classes (for example, the link between :Customer and :Payment in Figure 6.14). Sequence diagrams do not explicitly show links, although an underlying link can be assumed or the message could not be sent. Collaboration diagrams are also useful when you want to view the complete set of messages from the point of view of one object. This is valuable when you are preparing a state diagram (see Chapter 7), since the state diagram needs to know everything that can happen to a class of objects.

There are no hard and fast rules about whether to use a sequence or a collaboration diagram in any particular situation. Some people like to use sequence diagrams early in the development process, as their layout tends to be easier for users to follow, and collaboration diagrams later on since they map more clearly onto the class diagram, but in the end the type of diagram used is a matter of individual choice.

Model consistency

Interaction diagrams bring together many existing models and modelling elements: from the use case model, the use cases, the actors, use case scenarios and descriptions, from the class

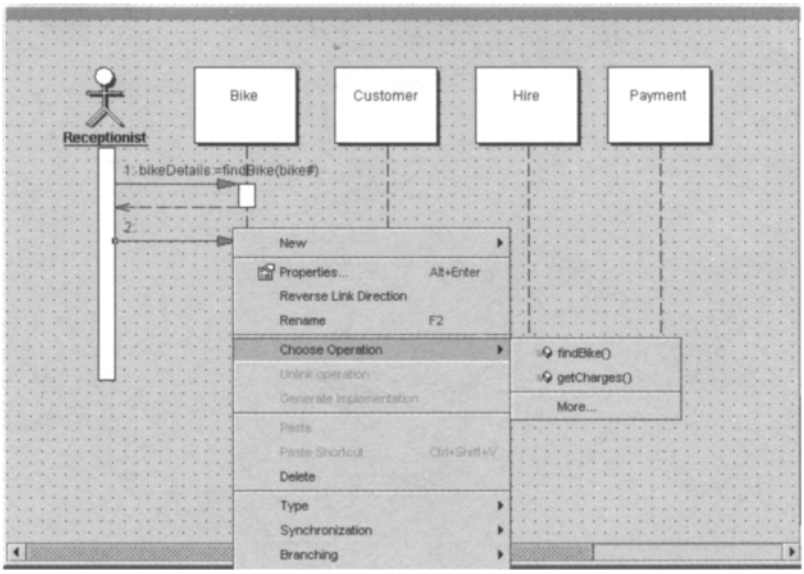

Figure 6.15 *Screen offering developer a choice of known operations for a message label*

diagram, the objects involved in each scenario and the operations on classes. A good CASE tool (see Chapter 1) will support model consistency by allowing the developer to link the objects on the interaction diagram to a list of classes it knows about from the class diagram. Similarly, it allows the developer to choose a label for the message arrow from a list of operations defined on the target object's class. Figure 6.15 shows a CASE tool offering a choice of operations (findBike() and getCharges()) for message number 2. The operations findBike() and getCharges() are defined on the class Bike in the class diagram.

Sequence diagrams are also useful for checking existing models; we may find, when doing the sequence diagrams, that we need an extra operation, or that we never use one that we did specify. A good CASE tool will allow us to add or delete operations and will update models (such as the class diagram) that are affected by our decision.

Using packages in interaction diagrams

On complicated interaction diagrams it is sometimes useful to suppress some of the details. The layout of collaboration diagrams makes it easy to identify groups of tightly coupled objects which can conveniently be regarded as a unit while we concentrate on what is happening in the rest of the diagram. In Figure 6.16a, objects e, f and g show complicated inter-object messaging. They can be grouped into a package and can be treated as a single entity while we concentrate on the interactions between objects a, b, c and d as in Figure 6.16b.

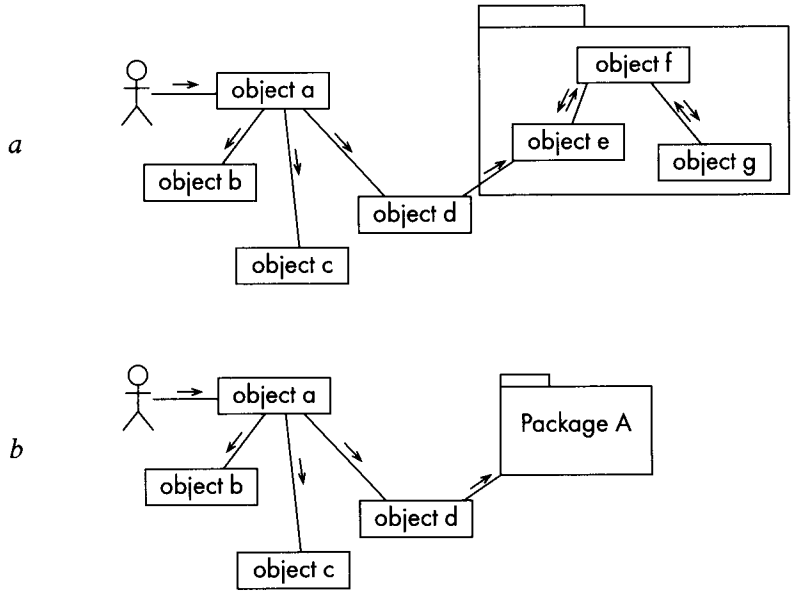

Figure 6.16 *a* *Grouping objects into a package in a collaboration diagram*
b *A package allows the developer to focus on object interactions in the rest of the diagram*

Specifying operations

In Chapter 5, we saw how a data dictionary notation can be used to document the details of the data in the developing system. However, that notation is restricted to data, and does not provide the means to record details about operations on classes. We have seen in this chapter that interaction diagrams are very useful for specifying the message passing between the group of objects involved in the execution of a use case scenario. However, these diagrams say very little about what happens inside an operation, they don't specify in any detail what an operation does. For this we need operation specifications.

Early on in the development of the system we are not concerned with the details of how an operation works; all we need at that stage is a brief description of what it does, not how it does it. For this sort of description the best tool to use is clear, everyday English or a mixture of English and data dictionary notation. For example, the operation findBike(bike#) in the Bike class can be described as follows:

findBike(bike#)

This operation finds the Bike object whose number corresponds to the bike number input (bike#) and returns

details about the bike (bike# + available + type + make + model + size + dailyHireRate + deposit)

We can see a more complex example in the description of the operation calcTotalPayment(amt, deposit) in the Payment class, as shown below:

calcTotalPayment(amt, deposit)

This operation calculates and records the sum of amounts paid as hire fees and the sum of deposits paid. This operation must find all current customer hire objects and for each one calculate the hire fee (Bike.dailyHireRate[1] * Hire.numberOfDays). The hire fees for all of the customer's hires are summed and recorded in Payment.totalAmountPaid. It also finds the deposit for each bike hired, sums them and records the result in Payment.totalDepositPaid.

As development progresses, we need to know more about how each operation carries out its processing. The UML does not provide any particular method for specifying operations, but activity diagrams (see Chapter 8) are an effective way of showing diagrammatically how an operation works.

An alternative approach, known as specification by contract, describes operations in terms of the services they deliver. This type of specification defines:

- The signature of the operation (its name, any arguments, and the type of values it returns)[2]
- The purpose of the operation
- What the client object must provide in order to obtain the required service
- A description of the internal logic of the operation
- Any other operations that are called by this operation
- Any attributes of objects whose values are changed by the operation.

1. *This notation means the attribute dailyHireRate in the class Bike.*
2. *This is a very important part of the specification because the signature of the operation is its public interface; as long as the signature remains unchanged, the internal details of the operation can be modified without affecting the rest of the system.*

As an example, we can use specification by contract to define the getCharges() operation (in the Bike class) which works out the cost of hiring a bike for a given number of days.

- getCharges(no.Days) : (deposit, dailyHireRate, total)
- This operation works out the cost of hiring a particular bike for a given number of days
- The bike details must have been found and the requested number of days of hire known
- The Bike object attribute dailyHireRate is multiplied by the number of days (no.Days). The result is added to the deposit to give the total. The operation returns the deposit, the dailyHireRate and the total
- This operation does not call any others
- This operation does not change the values of any attributes.

The internal logic of an operation can be described in a number of ways depending on the complexity of the algorithm involved. One of the most popular approaches is to use semi-formal, structured English. Structured English is a limited and structured subset of natural language, with a syntax that is similar to that of a block-structured programming language. Structured English generally includes the following constructs:

- A sequence construct:
  e.g. the second statement below is executed immediately after the first statement;
  Get bikeDetails[3]
  Get hireCharges
- Two decision constructs:
  e.g. IF customer is existing customer
  THEN confirm customerDetails
  ELSE record customerDetails
  or: CASE customer is existing customer, confirm customer details
- Two repetition constructs:
  e.g. WHILE more bikes to add DO enter bikeDetails
  or: REPEAT enter bikeDetails UNTIL no more bikes to add
- comments enclosed in parentheses;
  (* this is a comment *)

[3] *Nouns that are in the data dictionary are written as they appear there.*

As an example of structured English, we will specify the calcDaysOverdue() operation, following the informal description as shown below.

calcDaysOverdue()
This operation uses today's date from the system clock and the attributes Hire.startDate and Hire.numberDays to calculate whether the bike has been returned late and if so by how many days. It calculates the overdue amount (Bike.dailyHireRate multiplied by the number of days late) and records it by executing latenessDeduction(amt).

In structured English this operation could be specified as follows:

```
Add numberDays to startDate to give return date
number of days late = today's date – return date
IF return date > today's date
    THEN (*bike is overdue*)
        latenessDeduction = Bike.dailyHireRate * number of days late
    ELSE (*bike is not overdue*)
Display latenessDeduction
```

When an operation involves a number of decisions, it is often helpful to specify these using a decision table or decision tree. For example, in a more sophisticated bike hire system, the hire charges could depend on the number of bikes a customer has hired in the past year and the number of bikes in the current hire. Let us imagine that Wheels introduce discount hire rates as follows:

> If a customer has already hired at least five bikes during the past year they get a 15% discount, unless the current hire is for three bikes or more, in which case they get a 25% discount.

Figure 6.17 shows how this could be represented in a decision table.

In the decision table all the possible conditions are listed in the top left-hand quarter of the table and all the actions in the bottom left-hand quarter. The top right-hand corner of the table tabulates, in separate columns, all possible combinations of conditions (a dash indicates that the condition is currently irrelevant). This is the Rules section. The appropriate actions are shown below each rule, in the bottom right-hand corner of the table.

Figure 6.18 shows the same information as a decision tree.

Other methods of specifying complex operations include the Object Constraint Language (OCL), which is UML's formal language for specifying constraints on an object model. The Wheels system is not nearly large or complicated enough to justify using OCL, but you can find details about the language in Bennett *et al.*, (2002).

	Rules		
Conditions	1	2	3
Customer has hired ≥ 5 bikes during the past year	N	Y	Y
Current hire is for ≥ 3 bikes	–	N	Y
Actions			
No discount	X		
15% discount		X	
25% discount			X

Figure 6.17 *Example of a decision table showing hire discounts*

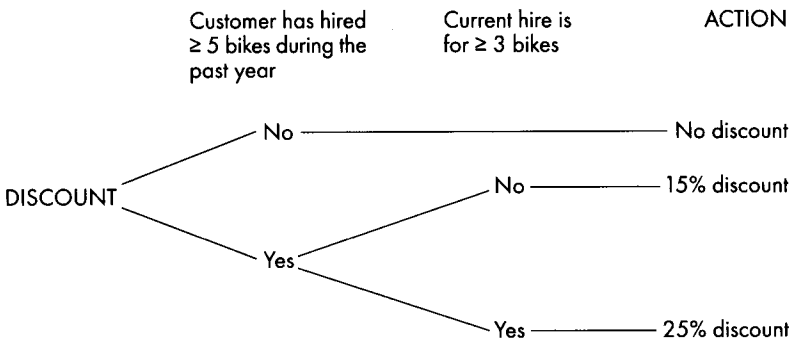

Figure 6.18 *Example of a decision tree showing hire discounts*

Using the CRC cards and interaction diagrams in system development

CRC cards are used to partition system behaviour between the classes. CRC modelling, by walking through the scenarios, has made us revisit all of our decisions so far – this is part of the iterative nature of object-oriented design. During CRC modelling we may well discover classes and attributes we didn't find doing a noun analysis because we are looking at the classes from a different point of view.

Each class of objects is responsible for some part of the system behaviour. However, for the system to produce a large chunk of required behaviour, for example that specified in a use case, objects must collaborate. The CRC technique is used to discover how classes collaborate to achieve the behaviour of the use cases. Interaction diagrams are used to document in detail the decision arrived at in CRC analysis. CRC cards talk about responsibilities and collaborations. Interaction diagrams talk about messaging between objects.

By the time we come to do the interaction diagrams, we have identified what the system will do and documented it in the use cases and use case scenarios. In the class diagram, we have identified the classes of objects that we think we will probably need, plus some of the attributes, and we have had a guess at what the relationships will be. A message from one object to another means that there should be association between the classes to which they belong. This is a useful check that we have got the associations right. If we find that there is no association on the class diagram between objects that need to message each other in an interaction diagram, then we need to amend the class diagram.

For a message from one object to another to work, the target object must understand the message. It can only do this if we have already defined that operation on its class. Interaction diagrams, therefore, act as a check that we have got the operations right.

Interaction diagrams can be shown at different levels of detail depending on when they are used during development. At their most detailed they can serve as comprehensive specifications of the use cases.

We would not expect an interaction diagram to be drawn for every possible scenario of every use case – a representative selection is enough. Generally these interaction diagrams model what normally happens in the successful execution of a use case (often referred to as the 'happy day' scenario) and the main alternative routes through, including ones where the use case goal is not achieved.

Technical points

Shading. On sequence diagrams we can shade activation boxes to give a more precise indication of when an object is *actively processing*, see Figure 6.19. This is different from when it is *active*. When :Customer sends a message to :Payment, it is still active, but it has passed control to :Payment. At this stage :Payment is doing the processing and :Customer, though still active, is just waiting for a response. :Customer cannot do any processing while it is waiting.

Here is a rather more complicated example that takes place when :Payment receives the calcTotalPayment() message.

- :Payment is active as soon as it receives the calcTotalPayment() message
- The calcTotalPayment() operation starts processing
- calcTotalPayment() stops processing when it sends the issueReceipt() message
- issueReceipt() is also an operation on :Payment, so a separate activation box is opened for the length of time that issueReceipt() is executing

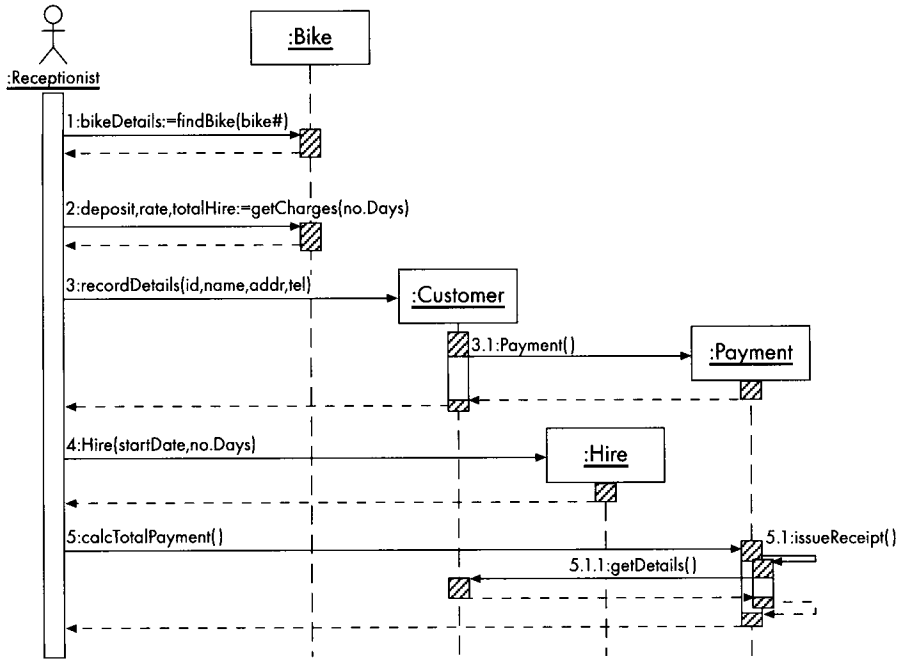

Figure 6.19 *Sequence diagram for the 'Issue bike' scenario with shading to show processing*

- issueReceipt() stops processing when it sends the getDetails() message to :Customer
- Senders regain control as the operations finish processing.

Iteration. The normal assumption on an interaction diagram is that the object icon at the top of a lifeline represents only one object. However, sometimes we want to send the same message to many objects. For example, in the 'Issue bike' use case, when the Receptionist is searching for the details of the bike with a specific bike number, the same message is sent to all of the bike objects until we get a match. This is indicated on the sequence diagram by the iteration marker (*), see Figure 6.20. We can also, optionally, specify how many times the message should be sent. The iteration clause, 'until bike# matched', is shown in square brackets after the asterisk. Bases for iteration are:

- The number of times the message is iterated, e.g. [i = 1..4]
- Repetition while an expression is true, e.g. [while more bikes]
- A for loop, e.g. [for all customers].

On the collaboration diagram, the UML uses a stacked icon to indicate a plurality of :Bike objects, see Figure 6.21. This is known as a *multiobject*.

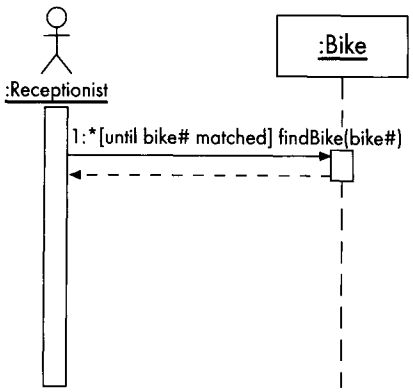

Figure 6.20 *Iteration marker*

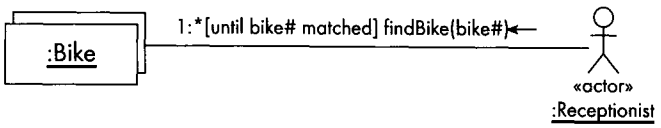

Figure 6.21 *A multiobject*

A multiobject indicates a collection of objects, which may be implemented as an array, a list, a set or some other data structure; the notation allows us to postpone this implementation decision. It is really a shorthand way of modelling the handling of a collection of objects by using a separate collection class. Collection classes are explained in Chapter 10.

Common problems

1 Can I show several actors participating in an interaction diagram? For instance, in the sequence diagram in Figure 6.22, I try to model the behaviour of the Customer and the Receptionist. I think there must be something wrong because the CASE tool would not allow me to call the second actor Customer, so I had to call it Client. I also had to add the messages between the Client and the Receptionist by hand.

There is no reason why you cannot have more than one actor in a sequence diagram, as long as they both participate in the use case and are genuinely inputting information to the system or receiving it directly from the system. However, CASE tools are very good indicators of errors – they understand how object-oriented models should work. Your model is trying to show an

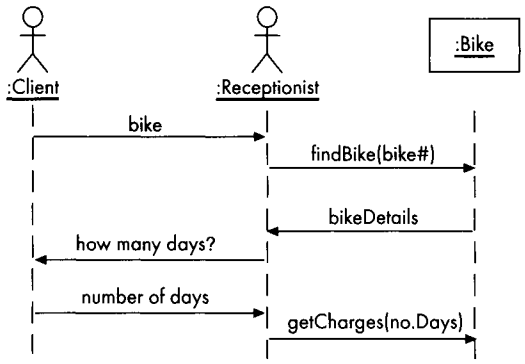

Figure 6.22 Incorrect modelling of the behaviour of two actors

interaction between the customer and the receptionist that is outside the scope of the system. We are not interested in what the customer and the receptionist say to each other, only in how they interact with the system.

The reason your CASE tool would not allow you to have an actor called Customer is probably that you have a Customer class on your class diagram. It is a very common modelling mistake to confuse the real customer with the electronic record of the customer's details.

Your diagram does make two other interesting points. First, it is quite valid to omit object activations. Second, the labels on the message arrows between the actors do not correspond to operations on classes and are therefore invalid. That is why you had to put them in by hand.

2 I can't get my CASE tool to automatically offer me a list of legitimate operations. I want to show that :Bike produces the bike details and the cost information. To get the CASE tool to do this I had to put all the message names in by hand. Is there something wrong with my diagram? See Figure 6.23.

:Bike does produce the items of information that you model, they are its outputs. However, what you have modelled are not messages but operation responses or returns. To get the CASE tool to work, you have to send the right message to :Bike to get it to execute whichever operation produces the outputs you want. Before that can happen you must have specified that this operation is an operation on the Bike class in the class diagram. Once you have done that, when you right-click your message arrow, most CASE tools will list the operations specified on the class of the target object so that you can select the one you want.

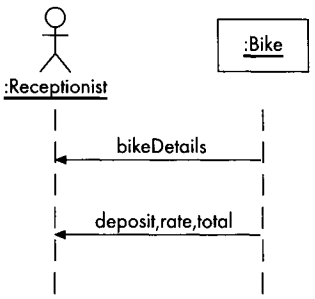

Figure 6.23 *Incorrect modelling of the messages between objects*

Chapter summary

This chapter focuses on the functionality of the system, which we started to look at in Chapter 3 on use cases. We show how to use CRC cards to identify responsibilities and allocate them between the classes in the system, and then discuss how to turn these responsibilities into operations. The two types of interaction diagram (sequence and collaboration) are introduced, and we illustrate how these diagrams model the message passing needed to achieve required functionality. We also look at informal, semi-formal and diagrammatic ways in which we can specify the details of operations. The Technical points and Common problems sections provide further information relating to sequence diagrams.

Bibliography

Bennett, S., McRobb, S. and Farmer, R. (2002) *Object-Oriented Systems Analysis and Design Using UML* (2nd edition), McGraw-Hill, London.

Britton, C. and Doake, J. (2000) *Object-Oriented Systems Development: A Gentle Introduction,* McGraw-Hill, London.

Fowler, M. (2000) *UML Distilled: A Brief Guide to the Standard Object Modeling Language* (2nd edition), Addison-Wesley, Reading, MA.

Lunn, K. (2003) *Software Development with UML,* Palgrave Macmillan, Basingstoke.

Priestly, M. (2000) *Practical Object-Oriented Design with UML,* McGraw-Hill, London.

Quatrani, T. (1998) *Visual Modeling with Rational Rose and UML*, Addison-Wesley, Reading, MA.

Stevens, P., with Pooley, R. (2000) *Using UML. Software Engineering with Objects and Components* (updated edition), Addison-Wesley, Harlow.

Wirfs-Brock, R., Wilkerson, B. and Wiener, L. (1990) *Designing Object-Oriented Software*, Prentice Hall, Englewood Cliffs, NJ.

Quick check questions

You can find the answers to these in the chapter.

a What aspect of a class is captured on a CRC card?

b How does a class deal with a responsibility that it cannot fulfil on its own?

c What are the two types of interaction diagram?

d What do interaction diagrams model?

e Apart from the format, what is the difference between a scenario and a sequence diagram?

f How are messages represented on sequence and collaboration diagrams?

g What does the thin rectangle on an object's lifeline indicate?

h List four ways in which collaboration diagrams differ from sequence diagrams.

i What features of an operation are defined in specification by contract?

j When is a decision tree or a decision table used to specify an operation?

Exercises

6.1 A mother is planning a birthday party for her son. She wants to invite his friends from school, but doesn't know where they all live, so she asks the form teacher for their addresses and phone numbers. She books a magician and buys lots of food and drink for the birthday tea. She makes the birthday cake herself, but has it iced by the local baker. Draw a CRC card to show the mother's responsibilities and whom she collaborates with to fulfil them.

Use case: Print ready card
Actors: Assistant
Goal: To print a postcard stating that a reserved video is available

Overview:
The Assistant uses the system to find details of the member who reserved the video. A postcard is printed with the member's name and address and the title of the video.

Typical course of events:

Actor action	System response
1 The Assistant asks for details of the reserving member	2 Displays details of the reserving member
3 The Assistant requests a printed postcard	4 Prints a postcard with name and address of reserving member and title of the video

Figure 6.24 *Use case description 'Print ready card'*

6.2 A mail order company is preparing a new brochure to send out to existing customers and other potential outlets. They will need to get an update on product details from their suppliers and a list of existing customers from the company database. The marketing department will supply them with a new design for the brochure and a list of people and organizations they can send it to. Draw a CRC card for the mail order company showing the responsibilities and collaborations in this situation.

6.3 Figure 6.24 shows the use case description 'Print ready card' from a video rental system (you can find more details about this system in the exercises in Chapter 3).

Following the guidelines given in the section on deriving operations from CRC responsibilities, identify two operations that are needed on the Reservation class.

6.4 Figure 6.25 shows the use case description 'Loan a video' from the video rental system.

Following the guidelines given in the section on deriving operations from CRC responsibilities, identify an operation that is needed on each of the following classes: Member, Video, Loan, and Payment.

Use case: Loan a video
Actors: Assistant
Goal: To lend a video to a customer

Overview:
A customer chooses a video and gives their membership card and the video to the Assistant. The Assistant scans the customer's membership card and checks if they owe any money or have outstanding loans. The system searches for a specific video using the barcode scanned from the video they wish to borrow. The system locates the required video and displays the details on the screen. The Assistant checks that this is the video the customer wants to borrow and looks to see what the rental cost is for this video. The system then registers the loan transaction.

Typical course of events:

Actor action	System response
1 The customer chooses a video	
2 The Assistant scans in the membership card barcode	3 Displays customer details
4 The Assistant agrees the details	
5 The Assistant scans in the video barcode	6 Displays video details including hire cost
7 The Assistant agrees the cost and registers the loan	8 Stores the loan transaction
9 The customer pays for the loan	
10 The Assistant records the payment	11 Prints a receipt

Figure 6.25 *Use case description 'Loan a video'*

6.5 Figure 6.26 shows the use case description 'Return a video' from the video rental system.

Following the guidelines given in the section on deriving operations from CRC responsibilities, identify two more operations that are needed on the Loan class and one on the Reservation class.

6.6 Describe the operation calcDaysOverdue() using specification by contract as described in the section on specifying operations. The informal description of this operation is as follows.

calcDaysOverdue()

This operation uses today's date from the system clock and the attributes Loan.startDate and Loan.numberDays

Use case: Return a video
Actors: Assistant
Goal: To register the return of a video by a customer

Overview:
A customer returns a video. The Assistant scans the video barcode and finds the record of the loan transaction. If the video is late back, the system will indicate this and a fine will be registered on the customer's record. The system then checks whether the returned title has been reserved by another member. If so the video is placed on the reserved shelf under the counter and a postcard is printed and sent to the reserving customer.

Typical course of events:

Actor action	System response
1 A customer returns a video	
2 The Assistant scans in the video barcode	3 Displays the loan transaction
4 The Assistant indicates that the video has been returned	5 Indicates if overdue
7 Assistant returns video to shelves	6 Checks for reservation

Figure 6.26 Use case description 'Return a video'

to calculate whether the video has been returned late and if so by how many days. It calculates the overdue amount (Video.dailyLoanRate multipled by the number of days late) and records it by executing setLatenessFine(amt).

6.7 A nationwide food and clothing store awards loyalty points to its customers in the following ways:

a If customers use the store's credit card in store, they get 4 points for every pound spent.

b If customers use the store's credit card at other outlets, they get 2 points for every pound spent.

c If customers shop in store, but do not use the store credit card, they get 1 point for every pound spent.

i Express this information in the form of a decision table.

ii Express the information in the form of a decision tree.

6.8 Figure 6.24 shows the use case description 'Print ready card' (to print a postcard stating that a reserved video is available) from a video rental system. You can find more details about

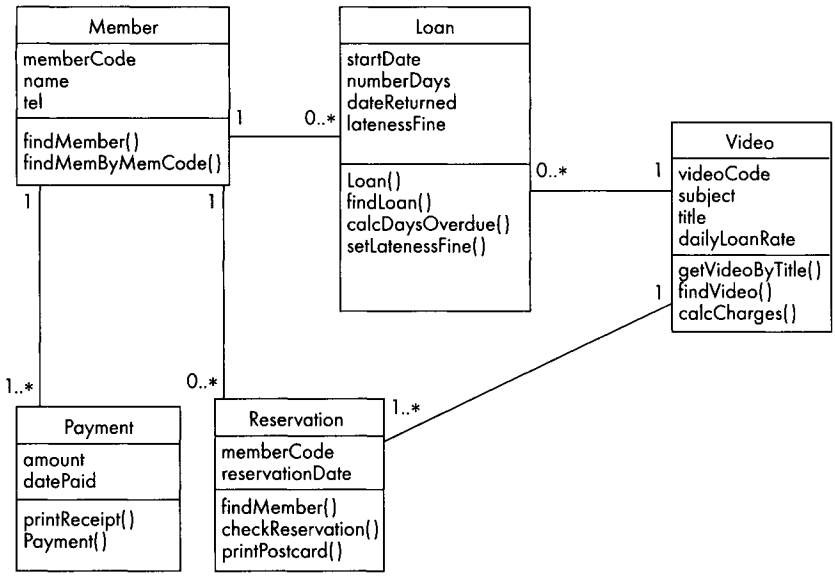

Figure 6.27 Class diagram for the video rental system

this system in the exercises in Chapter 3. Using the class diagram in Figure 6.27:

a Draw a sequence diagram for a successful scenario.

b Draw a collaboration diagram of the same scenario.

6.9 Figure 6.25 shows the use case description 'Loan a video' from the video rental system. Using the class diagram in Figure 6.27 draw a sequence diagram for a successful loan.

6.10 Figure 6.26 shows the use case description 'Return a video' from the video rental system. Using the class diagram in Figure 6.27 draw a sequence diagram for the late return of a video.