some discrete steps to take toward the ultimate goal and undertaking a process of iterative development.

16.4.1 Development steps

Planning some development steps helps us to consider how we might break up a single large problem into several smaller problems. Individually, these smaller problems are likely to be both less complex and more manageable than the one big problem, but together they should combine to form the whole. As we seek to solve the smaller problems, we might find that we need to also break up some of them. In addition, we might find that some of our original assumptions were wrong or that our design is inadequate in some way. This process of discovery, when combined with an iterative development approach, means that we obtain valuable feedback on our design and on the decisions we make, at an early enough stage for us to be able to incorporate it back into a flexible and evolving process.

Considering what steps to break the overall problem into has the added advantage of helping to identify some of the ways in which the various parts of the application are interconnected. In a large project, this process helps us to identify the interfaces between components. Identifying steps also helps in planning the timing of the development process.

It is important that each step in an iterative development represent a clearly identifiable point in the evolution of the application toward the overall requirements. In particular, we need to be able to determine when each step has been completed. Completion should be marked by the passing of a set of tests and a review of the step's achievements, so as to be able to incorporate any lessons learned into the steps that follow.

Here is a possible series of development steps for the taxi company application:

- Enable a single passenger to be picked up and taken to her destination by a single taxi.
- Provide sufficient taxis to enable multiple independent passengers to be picked up and taken to their destinations concurrently.
- Enable a single passenger to be picked up and taken to his destination by a single shuttle.
- Ensure that details are recorded of passengers for whom there is no free vehicle.
- Enable a single shuttle to pick up multiple passengers and carry them concurrently to their destinations.
- Provide a GUI to display the activities of all active vehicles and passengers within the simulation.
- Ensure that taxis and shuttles are able to operate concurrently.
- Provide all remaining functionality, including full statistical data.

We will not discuss the implementation of all of these steps in detail, but we will complete the application to a point where you should be able to add the remaining functionality yourself.

Exercise 16.17 Critically assess the list of steps we have outlined, with the following questions in mind. Do you feel the order is appropriate? Is the level of complexity of each too high, too low, or just right? Are there any steps missing? Revise the list as you see fit, to suit your own view of the project.

Exercise 16.18 Are the completion criteria (tests on completion) for each stage sufficiently obvious? If so, document some tests for each.

16.4.2 A first stage

For the first stage, we want to be able to create a single passenger, have them picked up by a single taxi, and have them delivered to their destination. This means that we shall have to work on a number of different classes: **`Location`**, **`Taxi`**, and **`TaxiCompany`**, for certain, and possibly others. In addition, we shall have to arrange for simulated time to pass as the taxi moves within the city. This suggests that we might be able to reuse some of the ideas involving actors that we saw in Chapter 12.

The *taxi-company-stage-one* project contains an implementation of the requirements of this first stage. The classes have been developed to the point where a taxi picks up and delivers a passenger to their destination. The **`run`** method of the **`Demo`** class plays out this scenario. However, more important at this stage are really the test classes—**`LocationTest`**, **`PassengerTest`**, **`PassengerSourceTest`**, and **`TaxiTest`**—which we discuss in Section 16.4.3.

Rather than discuss this project in detail, we shall simply describe here some of the issues that arose from its development out of the previous outline version. You should supplement this discussion with a thorough reading of the source code.

The goals of the first stage were deliberately set to be quite modest, yet still relevant to the fundamental activity of the application—collecting and delivering passengers. There were good reasons for this. By setting a modest goal, the task seemed achievable within a reasonably short time. By setting a relevant goal, the task was clearly taking us closer toward completing the overall project. Such factors help to keep our motivation high.

We have borrowed the concept of actors from the *foxes-and-rabbits* project of Chapter 12. For this stage, only taxis needed to be actors, through their **`Vehicle`** superclass. At each step a taxi either moves toward a target location or remains idle (Code 16.2). Although we did not have to record any statistics at this stage, it was simple and convenient to have vehicles record a count of the number of steps for which they are idle. This anticipated part of the work of one of the later stages.

The need to model movement required the **`Location`** class to be implemented more fully than in the outline. On the face of it, this should be a relatively simple container for a two-dimensional position within a rectangular grid. However, in practice, it also needs to provide both a test for coincidence of two locations (**`equals`**), and a way for a vehicle to find out where to move to next, based on its current location and its destination (**`nextLocation`**). At this stage, no limits were put on the grid area (other than that coordinate values should be positive), but this raises the need in a later stage for something to record the boundaries of the area in which the company operates.

Code 16.2
The **Taxi** class as an actor

```
/**
 * A taxi is able to carry a single passenger.
 *
 * @author David J. Barnes and Michael Kölling
 * @version 2016.02.29
 */

public class Taxi extends Vehicle
{
    private Passenger passenger;

    /**
     * Constructor for objects of class Taxi
     * @param company The taxi company. Must not be null.
     * @param location The vehicle's starting point. Must not be null.
     * @throws NullPointerException If company or location is null.
     */
    public Taxi(TaxiCompany company, Location location)
    {
        super(company, location);
    }

    /**
     * Carry out a taxi's actions.
     */
    public void act()
    {
        Location target = getTargetLocation();
        if(target != null) {
            // Find where to move to next.
            Location next = getLocation().nextLocation(target);
            setLocation(next);
            if(next.equals(target)) {
                if(passenger != null) {
                    notifyPassengerArrival(passenger);
                    offloadPassenger();
                }
                else {
                    notifyPickupArrival();
                }
            }
        }
        else {
            incrementIdleCount();
        }
    }

    /**
     * Is the taxi free?
     * @return Whether or not this taxi is free.
     */
    public boolean isFree()
    {
        return getTargetLocation() == null && passenger == null;
    }
```

Code 16.2 continued
The **Taxi** class as an actor

```
    /**
     * Is the taxi free?
     * @return Whether or not this taxi is free.
     */
    public boolean isFree()
    {
        return getTargetLocation() == null && passenger == null;
    }

    /**
     * Receive a pickup location. This becomes the
     * target location.
     * @param location The pickup location.
     */
    public void setPickupLocation(Location location)
    {
        setTargetLocation(location);
    }

    /**
     * Receive a passenger.
     * Set their destination as the target location.
     * @param passenger The passenger.
     */
    public void pickup(Passenger passenger)
    {
        this.passenger = passenger;
        setTargetLocation(passenger.getDestination());
    }

    /**
     * Offload the passenger.
     */
    public void offloadPassenger()
    {
        passenger = null;
        clearTargetLocation();
    }

    /**
     * Return details of the taxi, such as where it is.
     * @return A string representation of the taxi.
     */
    public String toString()
    {
        return "Taxi at " + getLocation();
    }
}
```

One of the major issues that had to be addressed was how to manage the association between a passenger and a vehicle, between the request for a pickup and the point of the vehicle's arrival. Although we were required only to handle a single taxi and a single passenger, we tried to bear in mind that ultimately there could be multiple pickup requests outstanding at any one time. In Section 16.2.3, we decided that a vehicle should receive its passenger

when it notifies the company that it has arrived at the pickup point. So, when a notification is received, the company needs to be able to work out which passenger has been assigned to that vehicle. The solution we chose was to have the company store a *vehicle:passenger* pairing in a map. When the vehicle notifies the company that it has arrived, the company passes the corresponding passenger to it. However, there are various reasons why this solution is not perfect, and we shall explore this issue further in the exercises below.

One error situation we addressed was that there might be no passenger found when a vehicle arrives at a pickup point. This would be the result of a programming error, so we defined the unchecked **`MissingPassengerException`** class.

As only a single passenger was required for this stage, development of the **`PassengerSource`** class was deferred to a later stage. Instead, passengers were created directly in the **`Demo`** and **`Test`** classes.

Exercise 16.19 If you have not already done so, take a thorough look through the implementation in the *taxi-company-stage-one* project. Ensure that you understand how movement of the taxi is effected through its **`act`** method.

Exercise 16.20 Do you feel that the **`TaxiCompany`** object should keep separate lists of those vehicles that are free and those that are not, to improve the efficiency of its scheduling? At what points would a vehicle move between the lists?

Exercise 16.21 The next planned stage of the implementation is to provide multiple taxis to carry multiple passengers concurrently. Review the **`TaxiCompany`** class with this goal in mind. Do you feel that it already supports this functionality? If not, what changes are required?

Exercise 16.22 Review the way in which *vehicle:passenger* associations are stored in the assignments' map in **`TaxiCompany`**. Can you see any weaknesses in this approach? Does it support more than one passenger being picked up from the same location? Could a vehicle ever need to have multiple associations recorded for it?

Exercise 16.23 If you see any problems with the current way in which *vehicle:passenger* associations are stored, would creating a unique identification for each association help—say a "booking number"? If so, would any of the existing method signatures in the **`Vehicle`** hierarchy need to be changed? Implement an improved version that supports the requirements of all existing scenarios.

16.4.3 Testing the first stage

As part of the implementation of the first stage, we have developed two test classes: **`LocationTest`** and **`TaxiTest`**. The first checks basic functionality of the **`Location`** class that is crucial to correct movement of vehicles. The second is designed to test that a passenger

is picked up and delivered to her destination in the correct number of steps, and that the taxi becomes free again immediately afterwards. In order to develop the second set of tests, the **`Location`** class was enhanced with the **`distance`** method, to provide the number of steps required to move between two locations.[1]

In normal operation, the application runs silently, and without a GUI there is no visual way to monitor the progress of a taxi. One approach would be to add print statements to the core methods of classes such as **`Taxi`** and **`TaxiCompany`**. However, BlueJ does offer the alternative of setting a breakpoint within the **`act`** method of, say, the **`Taxi`** class. This would make it possible to "observe" the movement of a taxi by inspection.

Having reached a reasonable level of confidence in the current state of the implementation, we have simply left print statements in the notification methods of **`TaxiCompany`** to provide a minimum of user feedback.

As testimony to the value of developing tests alongside implementation, it is worth recording that the existing test classes enabled us to identify and correct two serious errors in our code.

Exercise 16.24 Review the tests implemented in the test classes of *taxi-company-stage-one*. Should it be possible to use these as regression tests during the next stages, or would they require substantial changes?

Exercise 16.25 Implement additional tests and further test classes that you feel are necessary to increase your level of confidence in the current implementation. Fix any errors you discover in the process of doing this.

16.4.4 A later stage of development

It is not our intention to discuss in full the completion of the development of the taxi company application, as there would be little for you to gain from that. Instead, we shall briefly present the application at a later stage, and we encourage you to complete the rest from there.

This more advanced stage can be found in the *taxi-company-later-stage* project. It handles multiple taxis and passengers, and a GUI provides a progressive view of the movements of both (Figure 16.2). Here is an outline of some of the major developments in this version from the previous one.

- A **`Simulation`** class now manages the actors, much as it did in the *foxes-and-rabbits* **`project`**. The actors are the vehicles, the passenger source, and a GUI provided by the **`CityGUI`** class. After each step, the simulation pauses for a brief period so that the GUI does not change too quickly.
- The need for something like the **`City`** class was identified during development of stage one. The **`City`** object defines the dimensions of the city's grid and holds a collection of all the items of interest that are in the city—the vehicles and the passengers.

[1] We anticipate that this will have an extended use later in the development of the application as it should enable the company to schedule vehicles on the basis of which is closest to the pickup point.