Advanced Case Study

21

Outcomes:

By the end of this chapter you should be able to:

- *specify system requirements by developing a* ***use case model***;
- *annotate a* ***composition*** *association on a UML diagram*;
- *specify* ***enumerated types*** *in UML and implement them in Java*;
- *develop test cases from* ***behaviour specifications*** *found in the use case model*;
- *use the* `TabPane` *class to create an attractive user interface*;
- *add* ***tool tips*** *to JavaFX components.*

21.1 Introduction

You have covered quite a few advanced topics now in this second semester. In this chapter we are going to take stock of what you have learnt by developing an application that draws upon all these topics. We will make use of Java's package notation for bundling together related classes; we will implement interfaces; we will catch and throw exceptions; we will make use of the collection classes in the `java.util` package and we will store objects to file. We will also make use of many JavaFX components to develop an attractive graphical interface.

As with the case study we presented to you in the first semester, we will discuss the development of this application from the initial stage of requirements analysis, through to final implementation and testing stages. Along the way we will look at a few new concepts.

Q. Charatan and A. Kans, *Java in Two Semesters*, Texts in Computer Science,
https://doi.org/10.1007/978-3-319-99420-8_21

21.2 System Overview

The application that we will develop will keep track of planes using a particular airport. So as not to overcomplicate things, we will make a few assumptions:

- there will be no concept of *gates* for arrival and departure—passengers will be met at a runway on arrival and be sent to a runway on departure;
- planes entering airport airspace and requesting to land are either called into land on a free runway, or are told to join a queue of circling planes until a runway becomes available;
- once a plane departs from the airport it is removed from the system.

21.3 Requirements Analysis and Specification

Many techniques are used to determine system requirements. Among others, these include interviewing the client, sending out questionnaires to the client, reviewing any documentation if a current system already exists and observing people carrying out their work. A common way to document these requirements in UML is to develop a **use case model**. A use case model consists of **use case diagrams** and **behaviour specifications**.

A *use case diagram* is a simple way of recording the *roles* of different users within a system and the services that they require the system to deliver. The users (people or other systems) of a system are referred to as **actors** in use case diagrams and are drawn as simple stick characters. The roles these actors play in the system are used to annotate the stick character. The services they require are the so-called *use cases*. For example, in an ATM application an actor may be a customer and one of the use cases (services) required would be to withdraw cash. A very simple use case diagram for our application is given in Fig. 21.1.

Figure 21.1 depicts the actors in this application (air traffic controllers and information officers) and the services these actors require (registering a flight, listing arrivals and so on). Once a list of use cases has been identified, *behaviour specifications* are used to record their required functionality. A simple way of recording behaviour specifications is to give a simple textual description for each use case. Table 21.1 contains behaviour specifications for each use case given in Fig. 21.1. Note that the descriptions are always given from the users' point of view.

As the system develops, the use case descriptions may be modified as detailed requirements become uncovered. These descriptions will also be useful when testing the final application, as we will see later.

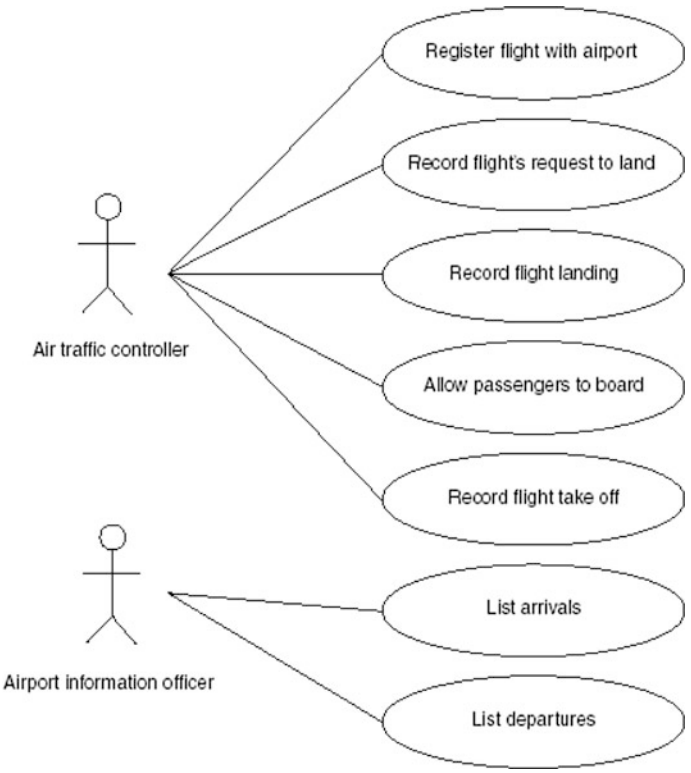

Fig. 21.1 A use case diagram for the airport application

Table 21.1 Behaviour specifications for the airport application

Register flight with airport	An air traffic controller registers an incoming flight with the airport by submitting its unique flight number, and its city of origin. If the flight number is already registered by the airport, the software will signal an error to the air traffic controller
Record flight's request to land	An air traffic controller records an incoming flight entering airport airspace, and requesting to land, by submitting its flight number. As long as the plane has previously registered with the airport, the air traffic controller is given an unoccupied runway number on which the plane will have permission to land. If all runways are occupied however, this permission is denied and the air traffic controller is informed to instruct the plane to circle the airport. If the plane has not previously registered with the airport, the software will signal an error to the air traffic controller
Record flight landing	An air traffic controller records a flight landing on a runway at the airport by submitting its flight number and the runway number. If the plane was not given permission to land on that runway, the software will signal an error to the air traffic controller

(continued)

Table 21.1 (continued)

Allow Passengers to board	An air traffic controller allows passengers to board a plane currently occupying a runway by submitting its flight number, and its destination city. If the given plane has not yet recorded landing at the airport, the software will signal an error to the air traffic controller
Record flight take off	An air traffic controller records a flight taking off from the airport by submitting its flight number. If there are planes circling the airport, the first plane to have joined the circling queue is then given permission to land on that runway. If the given plane was not at the airport, the software will signal an error to the air traffic controller
List arrivals	The airport information officer is given a list of planes whose status is either due-to-land, waiting-to-land, or landed
List departures	The airport information officer is given a list of planes whose status is currently waiting-to-depart (taking on passengers)

21.4 Design

The detailed design for this application is now presented in Fig. 21.2. It introduces some new UML notation. Have a look at it and then we will discuss it.

As you can see from Fig. 21.2, an `Airport` class has been introduced to represent the functionality of the system as a whole. The **`public`** methods of the `Airport` class correspond closely to the use cases identified during requirements analysis and specification. Notice we have provided two constructors. One that will allow us to create an empty `Airport` object and another that allows us to provide a filename (as a `String`) and load data stored in the given file. The **`private`** methods of the `Airport` class are there simply to help implement the functionality of the class.

The requirements made clear that there would be *many* planes to process in this system. Since the airport exists regardless of the number of planes at the airport, the relationship between the `Airport` and `Plane` class is one of containment, as indicated with a hollow diamond. It makes sense to consider the collection classes in the `java.util` package at this point. As we record planes in the system, and process these planes, we will always be using a plane's flight number as a way of identifying an individual plane. A `Map` is the obvious collection to choose here, with flight numbers the *keys* of the `Map` and the planes associated with these flight numbers as *values* of the `Map`.

The one drawback with a `Map`, however, is that it is not ordered on input. When considering which plane in a circling queue of planes to land, ordering is important, as the first to join the queue should be the first to land. So we have also introduced a `List` to hold the flight numbers of circling planes. Notice that the contained `Plane` type requires `equals` and `hashCode` methods to work effectively with these collection classes.

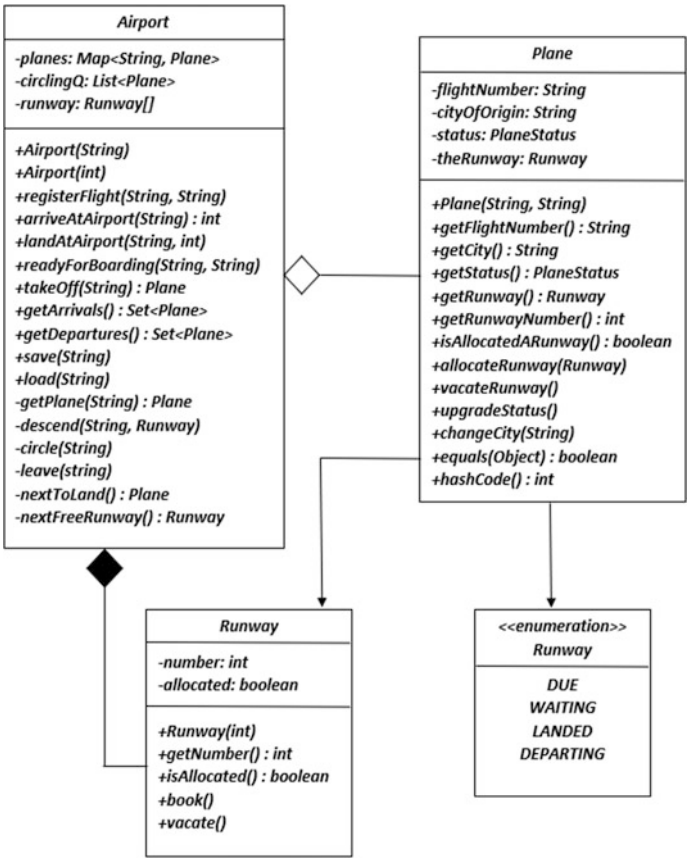

Fig. 21.2 Detailed design for the airport application

The airport will also consist of a number of runways. In fact the airport cannot exist without this collection of runways. The airport is said to be *composed of* a number of runways as opposed to *containing* a number of planes. Notice that the UML notation for **composition** is the same as that for containment, except that the diamond is filled rather than hollow. We use an array to hold this collection of `Runway` objects.

Turning to the contained classes, the `Runway` class provides methods to allow for the runway number to be retrieved, and for a runway to be booked and vacated. The `Plane` class also has access to a `Runway` object, to allow a plane to be able to book and vacate runways. You can see that as well as each plane being associated with a runway, the plane also has a flight number, a city and a status associated with it. The arrows from the `Plane` class to the `PlaneStatus` and `Runway` classes indicate the direction of the association. In this case a `Plane` object can send messages to a `Runway` and `PlaneStatus` object, but not vice versa.

The status of a plane is described in the `PlaneStatus` diagram. This diagram is the UML notation for an *enumerated type*, which is a type we have not met before.

21.5 Enumerated Types in UML

A type that consists of a few possible values, each with a meaningful name, is referred to as an **enumerated type**. The status of a plane is one example of an enumerated type. This status changes depending upon the plane's progress to and from the airport:

- when a plane registers with the airport, it is *due* to land;
- when a plane arrives in the airport's airspace, it is *waiting* to land (this plane may be told to come in and land, or it may have to circle the airport until a runway becomes available);
- when a plane touches down at the airport, it has *landed*;
- when a plane starts boarding new passengers, it is *departing* the airport.

You can see from the design of the system that such a type is captured in UML by marking this type with `<<enumeration>>` as follows (Fig. 21.3).

We need to mark this UML diagram with `<<enumeration>>` so that it is not confused with a normal UML class diagram. With a normal UML class diagram, attributes and methods are listed in the lower portion. With an enumerated type diagram, the possible values of this type are given in the lower portion of the diagram, with each value being given a meaningful name. An attribute that is allocated a `PlaneStatus` type, such as `status` in the `Plane` class, can have any one of these values.

This completes our design analysis, so now let's turn our attention to the Java implementation.

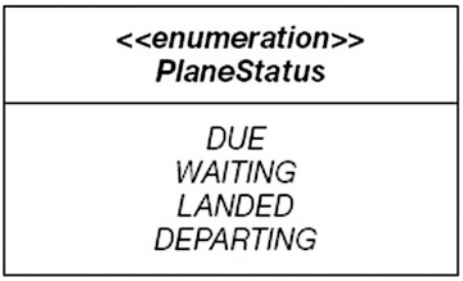

Fig. 21.3 The UML design of the enumerated *PlaneStatus* type

21.6 Implementation

Since we are developing an application involving several classes, it makes sense to bundle these classes together into a single package. We will call this package `airportSys`. This means that all our classes will begin with the following **`package`** statement:

```
package airportSys;
```

It is a good idea to hide implementation level exceptions (such as `NumberFormatException`) from users of the application and, instead, always throw some general application exception. In order to be able to do this, we define our own general `AirportException` class.

AirportException

```
package airportSys; // add to package

/**
 * Application Specific Exception
 *
 * @author Charatan and Kans
 * @version 1st August 2018
 */
public class AirportException extends RuntimeException
{
 /**
  * Default Constructor
  */
 public AirportException ()
 {
    super("Error: Airport System Violation");
 }

 /**
  * Constructor that accepts an error message
  */
 public AirportException (String msg)
 {
    super(msg);
 }
}
```

Notice that, as with all the classes we shall develop here, we have added `Javadoc` comments into the class definition. Now let's consider the remaining classes. First of all, we will look at the implementation of the enumerated `PlaneStatus` type.

21.6.1 Implementing Enumerated Types in Java

In order to define an enumerated type such as `PlaneStatus`, the **`enum`** keyword is used. The `PlaneStatus` type can now be implemented simply as follows:

```
// this is how to definine an enumerated type in Java
public enum PlaneStatus
{
    DUE, WAITING, LANDED, DEPARTING
}
```

You can see how easy it is to define an enumerated type. When defining such a type, do not use the **`class`** keyword, use the **`enum`** keyword instead. The different values for this type are then given within the braces, separated by commas.

These values create class constants, with the given names, as before. The type of each class constant is `PlaneStatus` and variables can now be declared of this type. For example, here we declare a variable of the `PlaneStatus` type and assign it one of these class constant values:

```
PlaneStatus status; // declare PlaneStatus variable
status = PlaneStatus.DEPARTING; // assign variable a class constant
```

The variable `status` can take no other values, apart from those defined in the enumerated `PlaneStatus` type. Each enumerated type you define will also have an appropriate `toString` method generated for it, so values can be displayed on the screen:

```
System.out.println("Value = " + status);
```

Assuming we created this variable as above, this would display the following:

`Value = DEPARTING`

As well as a `toString` method, a few other methods are generated for you as well, and the **`switch`** statement can be used in conjunction with enumerated type variables. We will see examples of these features when we look at the code for the other classes in this application.

Of course, we must remember to add this `PlaneStatus` type into our `airportSys` package:

The *PlaneStatus* type

```
package airportSys; // add to package

/**
 * Enumerated plane status type.
 *
 * @author Charatan and Kans
 * @version 1st August 2018
 */
public enum PlaneStatus
{
    DUE, WAITING, LANDED, DEPARTING
}
```

21.6.2 The *Runway* Class

Here is the code for the `Runway` class, take a look at it and then we will discuss it.

Runway

```
package airportSys; // add class to package
import java.io.Serializable;

/**
 * This class is used to store details of a single runway.
 *
 * @author Charatan and Kans
 * @version 1st August 2018
 */
public class Runway implements Serializable
{
 // attributes
 private int number;
 private boolean allocated;

 /**
  * Constructor sets the runway number
  * @param       numberIn              Used to set the runway number
  * @throws      AirportException      When the runway number is less than 1
  */
 public Runway (int numberIn)
 {
   if (numberIn <1)
   {
      throw new AirportException ("invalid runway number "+numberIn);
   }
    number = numberIn;
    allocated = false; // runway vacant initially
 }

 /**
  * Returns the runway number
  */
 public int getNumber()
 {
    return number;
 }

 /**
  * Checks if the runway has been allocated
  * @return      Returns true if the runway has been allocated and false otherwise
  */
 public boolean isAllocated()
 {
    return allocated;
 }

 /**
  * Records the runway as being booked
  */
 public void book()
 {
   allocated = true;
 }

 /**
  * Records the runway as being vacant
  */
 public void vacate()
 {
    allocated = false;
 }
}
```

There is not much that needs to be said about this class. As we may wish to save and load objects from our system, we have to remember to indicate that this class is `Serializable`.

```
public class Runway implements Serializable
```

Notice that we have defined this as a **public** class so that it is accessible outside of the package. We did this as a runway is a generally useful concept in many applications; declaring this class **public** allows it to be re-used outside of the `airportSys` package. In fact, we have declared most of our classes **public** for this reason.

21.6.3 The *Plane* Class

Here is the code for the `Plane` class. Have a close look at it and then we will discuss it.

Plane

```
package airportSys;

import java.io.Serializable;

/**
 * This class stores the details of a single plane
 *
 * @author Charatan and Kans
 * @version 2nd August 2018
 */
public class Plane implements Serializable
{

  // attributes
  private String flightNumber;
  private String city;
  private PlaneStatus status;
  private Runway theRunway; // to implement Runway association

  // methods

  /**
   * Constructor sets initial flight details of the plane requesting registration
   *
   * @param   flightIn      The flight number of the plane to register
   * @param   cityOfOrigin  The city of origin of the plane to register
   */
  public Plane(String flightIn, String cityOfOrigin)
  {
    flightNumber = flightIn;
    city = cityOfOrigin;
    status = PlaneStatus.DUE; // initial plane status set to DUE
    theRunway = null; // indicates no runway allocated
  }

  /**
   * Returns the plane's flight number
   */
  public String getFlightNumber()
  {
    return flightNumber;
  }

  /**
   * Returns the city associated with the flight
   */
  public String getCity()
  {
    return city;
  }

  /**
   * Returns the current status of the plane
   */
  public PlaneStatus getStatus()
  {
    return  status;
  }

  /**
   * Returns the runway allocated to this plane or null if no runway allocated
   */
  public Runway getRunway()
```

```
  {
    return  theRunway;
  }

  /**
   * Returns the runway number allocated to this plane
   * @throws    AirportException if no runway allocated
   */
  public int getRunwayNumber()
  {
    if (theRunway == null)
    {
      throw new AirportException ("flight "+flightNumber+" has not been allocated a runway");
    }
    return theRunway.getNumber();
  }

  /**
   * Checks if the plane is allocated a runway
   * @return    Returns true if the plane has been allocated a runway
   *            and false otherwise
   */
  public boolean isAllocatedARunway()
  {
    return theRunway!=null;
  }

  /**
   * Allocates the given runway to the plane
   *
   * @throws     AirportException if runway parameter is null or runway already allocated
   */
  public void allocateRunway(Runway runwayIn)throws AirportException
  {
    if (runwayIn == null) // check runway has been sent
    {
      throw new AirportException ("no runway to allocate");
    }
    if (runwayIn.isAllocated())
    {
        throw new AirportException ("runway already allocate");
    }
    theRunway = runwayIn;
    theRunway.book();
  }

  /**
   * De-allocates the current runway
   *
   * @throws    AirportException if no runway allocated
   */
  public void vacateRunway()
  {
    if (theRunway==null)
    {
       throw new AirportException ("no runway allocated");
    }
    theRunway.vacate();
  }

  /**
   * Returns the String representation of the plane's status
   */
  public String getStatusName()
  {
      return status.toString();
  }


  /**
   * Upgrades the status of the plane.
   */
  public void upgradeStatus()
  {
    switch(status)
    {
       case DUE: status =PlaneStatus.WAITING; break;
       case WAITING: status =PlaneStatus.LANDED; break;
       case LANDED: status =PlaneStatus.DEPARTING; break;
       case DEPARTING: throw new AirportException("Cannot upgrade DEPARTING status");
    }
  }

  /**
   * Changes the city associated with the plane
   */
  public void changeCity (String destination)
```

```
  {
    city = destination;
  }

  /**
  * Returns a string representation of a plane
  */
  @Override
  public String toString()
  {
    String out =    "number: "+flightNumber+   "\tcity: "+city+ "\tstatus: "+status;
    if (theRunway!=null)
    {
            out = out +"\trunway: "+theRunway;
    }
    return out;
  }


  /**
   * Checks whether the plane is equal to the given object
   */
  @Override
  public boolean equals(Object objIn)
  {
    if (objIn!=null)
    {
           Plane p = (Plane)objIn;
           return p.flightNumber.equals(flightNumber);
    }
    else
    {
           return false;
    }
 }

  /**
   * Returns a hashcode value
   */
  @Override
  public int hashCode()
  {
    return flightNumber.hashCode();
  }
}
```

Again, most of the points we raised with the `Runway` class are relevant to this `Plane` class. It needs to be `Serializable` and it is declared **`public`**.

Since `Plane` objects will be used in collection classes we have provided this class with an `equals` and a `hashCode` method. You can see that both of these methods make use of the plane's flight number.

In addition you should look at the way in which we dealt with the `status` attribute. During class design we declared this attribute to be of the enumerated `PlaneStatus` type, so it has been implemented as follows:

```
private PlaneStatus status;
```

We can then assign this attribute values from the enumerated `PlaneStatus` type. For example, in the constructor, we initialize the status of a plane to `DUE`:

```
public Plane(String flightNumberIn, String cityOfOrigin)
{
   flightNumber = flightNumberIn;
   city = cityOfOrigin;
   status = PlaneStatus.DUE;
   theRunway = null;
}
```

The `getStatus` method returns the value of the `status` attribute, so the appropriate return type is `PlaneStatus`:

```
public PlaneStatus getStatus()
{
    return status;
}
```

The `upgradeStatus` method is interesting as it demonstrates how the **`switch`** statement can be used with enumerated type variables such as `status`:

```
public void upgradeStatus()
{
    switch(status) // this is an enumerated type variable
    {
        // 'case' statements can check the different enumerated values
        case DUE: status = PlaneStatus.WAITING; break;
        case WAITING: status = PlaneStatus.LANDED; break;
        case LANDED: status = PlaneStatus.DEPARTING; break;
        case DEPARTING: throw new AirportException("Cannot upgrade DEPARTING status");
    }
}
```

Here we are upgrading the status of a plane as it makes its way to, and eventually from, the airport. Notice that the value of the `status` attribute is checked in the **`case`** statements, but this value is *not* appended onto the `PlaneStatus` class name. For example:

```
// just use a status name in 'case' test
case DUE: status = PlaneStatus.WAITING; break;
```

However, in all other circumstances, such as assigning to the `status` attribute, the enumerated value *does* have to be appended onto the `PlaneStatus` class name:

```
// use class + status name in all other circumstances
case DUE: status = PlaneStatus.WAITING; break;
```

You can see that we should not be upgrading the status of a plane if its current status is `DEPARTING`, so an exception is thrown in this case:

```
case DEPARTING: throw new AirportException ("Cannot upgrade DEPARTING status");
```

Before we leave this class, also notice that by adding a runway attribute, `theRunway`, into the `Plane` class we can send messages to (access methods of) a `Runway` object, for example:

```
public void allocateRunway(Runway runwayIn)
{
    // some code here
    theRunway.book(); // 'book' is a 'Runway' method
}
```

21.6.4 The *Airport* Class

The `Airport` class encapsulates the functionality of the system. It does not include the interface to the application. As we have done throughout this book, the interface of an application is kept separate from its functionality. That way, we can modify the way we choose to implement the functionality without needing to modify the interface, and vice versa. Examine it closely, being sure to read the comments, and then we will discuss it.

Airport

```
package airportSys;

import java.util.Map;
import java.util.HashMap;
import java.util.List;
import java.util.ArrayList;
import java.util.Set;
import java.util.HashSet;
import java.io.IOException;
import java.io.FileOutputStream;
import java.io.ObjectOutputStream;
import java.io.FileInputStream;
import java.io.ObjectInputStream;

/**
 * Class to provide the functionality of the airport system
 *
 * @author Charatan and Kans
 * @version 4th August 2018
 */
public class Airport
{

  // attributes
  private Map<String, Plane> planes;  // registered planes
  private List<String> circlingQ; //flight numbers of circling planes
  private Runway []runway; // runways allocated to the airport

  // methods

  /**
   * This Constructor allows airport data to be loaded from a file
   *
   * @param   filenameIn The name of the file
   * @throws  IOException if problems with opening and loading given file
   * @throws  ClassNotFoundException if objects in file not of the right type
   */
  public Airport(String filenameIn)throws IOException, ClassNotFoundException
  {
      load(filenameIn);
  }


  /**
   * This Constructor creates an empty collection of planes,
   * and allocates runways to the airport
   *
   * @param   numIn The number of runways
   * @throws  AirportException if negative runway number used
   */
  public Airport (int numIn)
  {
    try
    {
        // intialise runways
        runway = new Runway [numIn];
        for (int i = 0; i<numIn; i++)
        {
          runway[i] = new Runway (i+1);
        }
        // initially no planes allocated to airport
        planes = new HashMap<>();
        circlingQ = new ArrayList<>();
    }
    catch (Exception e)
    {
        // notice throwing an excpetion from a catch clause
        throw new AirportException("Invalid Runway Number set, application closing");
    }
  }

  /**
   * Registers a plane with the airport
   *
   * @param   flightIn The plane's flight number
```

```
 * @param   cityOfOrigin The plane's city of origin
 * @throws  AirportException if flight number already registered.
 */
public void registerFlight (String flightIn, String cityOfOrigin)
{
   if (planes.containsKey(flightIn))
   {
      throw new AirportException ("flight "+flightIn+" already registered");
   }
   Plane newPlane = new Plane (flightIn, cityOfOrigin);
   planes.put(flightIn, newPlane);
}

 /**
 * Records a plane arriving at the airport
 *
 * @param   flightIn The plane's flight number
 * @throws  AirportException if plane not previously registered
 *            or if plane already arrived at airport
 */
public int arriveAtAirport (String flightIn)
{
   Runway vacantRunway = nextFreeRunway(); // get next free runway
   if (vacantRunway != null) // check if runway available
   {
      descend(flightIn, vacantRunway); // allow plane to descend on this runway
      return vacantRunway.getNumber(); // return booked runway number
   }
   else // no runway available
   {
      circle(flightIn); // plane must join circling queue
      return 0; // indicates no runway available to land
   }
}

/**
 * Records a plane landing on a runway
 *
 * @param   flightIn The plane's flight number
 * @param   runwayNumberIn The runway number the plane is landing on
 * @throws  AirportException if plane not previously registered
 *                           or if the runway is not allocated to this plane
 *                           or if plane has not yet signalled its arrival at the aiport
 *                           or if plane is already recorded as having landed.
 */
public void landAtAirport (String flightIn, int runwayNumberIn)
{
  Plane thisPlane = getPlane(flightIn); // throws AirportException if not registered
  if (thisPlane.getRunwayNumber()!= runwayNumberIn)
  {
    throw new AirportException ("flight "+flightIn+" should not be on this runway");
  }
  if (thisPlane.getStatus()== PlaneStatus.DUE)
  {
    throw new AirportException ("flight "+flightIn+" not signalled its arrival");
  }
  if (thisPlane.getStatus().compareTo(PlaneStatus.WAITING)>0)
  {
    throw new AirportException ("flight "+flightIn+" already landed");
  }
  thisPlane.upgradeStatus(); // upgrade status from WAITING to LANDED
}

/**
 * Records a plane boarding for take off
 *
 * @param   flightIn The plane's flight number
 * @param   destination The city of destination
 * @throws  AirportException  if plane not previously registered
 *                            or if plane not yet recorded as landed
 *                            or if plane already recorded as ready for take off
 */
public void readyForBoarding(String flightIn, String destination)
{
  Plane thisPlane = getPlane(flightIn);// throws AirportException if not registered
  if (thisPlane.getStatus().compareTo(PlaneStatus.LANDED)<0)
  {
    throw new AirportException ("flight "+flightIn+" not landed");
  }
  if (thisPlane.getStatus()== PlaneStatus.DEPARTING)
  {
    throw new AirportException ("flight "+flightIn+" already registered to depart");
```

```
    }
    thisPlane.upgradeStatus(); // upgrade status from LANDED to DEPARTING
    thisPlane.changeCity(destination); // change city of origin to city of destination
  }

  /**
   * Records a plane taking off from the aiprort
   *
   * @param   flightIn The plane's flight number
   * @throws  AirportException     if plane not previously registered
   *                               or if plane not yet recorded as landed
   *                               or if the plane not previously recorded as taken off
   */
  public Plane takeOff (String flightIn)
  {
    leave(flightIn); // remove from plane register
    Plane nextFlight = nextToLand(); // return next circling plane to land
    if (nextFlight != null) // check circling flight exists
    {
      Runway vacantRunway = nextFreeRunway();
      descend(nextFlight.getFlightNumber(), vacantRunway); // allocate runway to circling plane
      return nextFlight; // send back details of next plane to land
    }
    else // no circling planes
    {
      return null;
    }
  }


  /**
   * Returns the set of planes due for arrival
   */
  public Set<Plane> getArrivals()
  {
      Set<Plane> planesOut = new HashSet<>();
      Set<String> items = planes.keySet();
      for(String thisFlight: items)
      {
        Plane thisPlane = planes.get(thisFlight);
        if (thisPlane.getStatus() != PlaneStatus.DEPARTING)
        {
              planesOut.add(thisPlane); //add to set
        }
      }
      return planesOut;
   }

  /**
   * Returns the set of planes due for departure
   */
  public Set<Plane> getDepartures()
  {
      Set<Plane> planesOut = new HashSet<>();
      Set<String> items = planes.keySet();
      for(String thisFlight: items)
      {
        Plane thisPlane = planes.get(thisFlight);
        if (thisPlane.getStatus()==PlaneStatus.DEPARTING)
        {
              planesOut.add(thisPlane); // add to set
        }
      }
      return planesOut;
  }

  /**
   * Returns the number of runways
   */
  public int getNumberOfRunways()
  {
    return runway.length;
  }

  /**
   * Saves airport object to file
   *
   * @param  fileIn The name of the file
   * @throws IOException if problems with opening and saving to given file
   */
  public void save(String fileIn)throws IOException
  {
      // notice try-with-resources to ensure file closes safely
      try ( FileOutputStream fileOut = new FileOutputStream(fileIn);
            ObjectOutputStream objOut = new ObjectOutputStream (fileOut))
      {
          objOut.writeObject(planes);
```

```
        objOut.writeObject(circlingQ);
        objOut.writeObject(runway);
    }
}

/**
 * Loads airport object from file
 *
 * @param  fileName The name of the file
 * @throws IOException if problems with opening and loading given file
 * @throws ClassNotFoundException if objects in file not of the right type
 */
public void load (String fileName) throws IOException, ClassNotFoundException
{
    // notice try-with-resources to ensure file closes safely
    try ( FileInputStream fileInput = new FileInputStream(fileName);
          ObjectInputStream objInput = new ObjectInputStream (fileInput))
    {
        planes = (Map<String, Plane>) objInput.readObject();
        circlingQ = (List<String>) objInput.readObject();
        runway = (Runway[])objInput.readObject();
    }
}


 // helper method to find next free runway
 private Runway nextFreeRunway()
 {
    for (Runway nextRunway : runway)
    {
        if (!nextRunway.isAllocated())
        {
            return nextRunway;
        }
    }
    return null;
 }

/**
 * Returns the registered plane with the given flight number
 *
 * @throws AirportException if flight number not yet registered.
 */
 private Plane getPlane(String flightIn)
 {
     if (!planes.containsKey(flightIn))
     {
        throw new AirportException ("flight "+flightIn+" has not yet registered");
     }
     return planes.get(flightIn);
  }

/**
 * Records a plane descending on a runway
 *
 * @param   flightIn The plane's flight number
 * @param   runwayIn The runway the plane will be landing on
 * @throws  AirportException    if plane not previously registered
 *                              or if plane already arrived at airport
 *                              or if plane already allocated a runway
 */
 private void descend (String flightIn, Runway runwayIn)
 {
  Plane thisPlane = getPlane(flightIn);// throws AirportException if not registered
  if (thisPlane.getStatus().compareTo(PlaneStatus.WAITING)>0)

  {
      throw new AirportException
        ("flight "+flightIn+" already at airport has status of "+thisPlane.getStatusName());
  }
  if (thisPlane.isAllocatedARunway())
  {
     throw new AirportException
        ("flight "+flightIn+" has already been allocated runway "+thisPlane.getRunwayNumber());
  }
  thisPlane.allocateRunway(runwayIn);
  if (thisPlane.getStatus()==PlaneStatus.DUE) // updraged status from DUE to WAITING
  {
      thisPlane.upgradeStatus();
  }
}

/**
 * Records a plane joining the planes circling the airport
 *
 * @param  flightIn The plane's flight number
 * @throws AirportException if plane not previously registered
```

```
   *                          or if plane already arrived
   */
  private void circle (String flightIn)
  {
    Plane thisPlane = getPlane(flightIn); // throws AirportException if not registered
    if (thisPlane.getStatus()!= PlaneStatus.DUE)
    {
      throw new AirportException ("flight "+flightIn+" already at airport");
    }
    thisPlane.upgradeStatus(); // updraged status from DUE to WAITING
    circlingQ.add(flightIn);
  }


  /**
   * Records a plane taking off from the aiprort
   *
   * @param  flightIn The plane's flight number
   * @throws  AirportException if plane not plane not not previously registered
   *                 or if plane not yet recorded as landed
   *                 or if the plane has not previously been recorded as ready for take off
   */
  private void leave (String flightIn)
  {
    // get plane associated with given flight number
    Plane thisPlane = getPlane(flightIn);// throws AirportException if not registered
    // throw exceptions if plane is not ready to leave airport
    if (thisPlane.getStatus().compareTo(PlaneStatus.LANDED)<0)
    {
      throw new AirportException ("flight "+flightIn+" not yet landed");
    }
    if (thisPlane.getStatus()==PlaneStatus.LANDED)
    {
       throw new AirportException ("flight "+flightIn+" must register to board");
    }
    // process plane leaving airport
    thisPlane.vacateRunway(); // runway now free
    planes.remove(flightIn); // remove plane from list
  }

  /**
   * Locates next circling plane to land
   *
   * @return     Returns the next circling plane to land
   *             or null if no planes
   */
  private Plane nextToLand()
  {
    if (!circlingQ.isEmpty()) // check circling plane exists
    {
      String flight =  circlingQ.get(0);
      circlingQ.remove(flight);
      return getPlane(flight); // could throw exception of not in list
    }
    else // no circling plane
    {
      return null;
    }
  }

}
```

There is not a lot that is new here, but we draw your attention to a few implementation issues.

First, notice we have provided two constructors, as specified in Fig. 21.2. The first receives the name of a file and loads data from this given file (using the **`private`** `load` method to be found later in this `Airport` class); the associated exceptions are passed on if an error occurs during this process:

```
public Airport(String filenameIn)throws IOException, ClassNotFoundException
{
      load(filenameIn); // call private method to load airport data
}
```

The second constructor receives the number of runways associated with this airport and initialises all data to be empty.

```
public Airport (int numIn)
{
  try
  {
      // intialise runways
      runway = new Runway [numIn];
      for (int i = 0; i<numIn; i++)
      {
        runway[i] = new Runway (i+1);
      }
      // initially no planes allocated to airport
      planes = new HashMap<>();
      circlingQ = new ArrayList<>();
  }
  catch (Exception e)
  {
      // notice we have thrown our user-defined exception from this catch clause
      throw new AirportException("Invalid Runway Number set, application closing");
  }
}
```

Within this constructor you can see that we catch a general exception, in case something goes wrong when allocating the array, and throw our application-specific exception when this occurs. This can be useful if we wish to suppress the name of Java specific expectations and stick to the names of our own user-defined exceptions.

Most of the other **`public`** methods simply check for a list of exceptions, and then upgrade the plane's status as it makes its way to and eventually from the airport.

Here, for example, is the method that records a plane that has previously landed at the airport, being ready to board new passengers for a new destination:

```
/**
   * Records a plane boarding for take off
   *
   * @param   flightIn The plane's flight number
   * @param   destination The city of destination
   * @throws  AirportException if plane not previously registered
   *                           or if plane not yet recorded as landed
   *                           or if plane already recorded as ready for take off
   */
  public void readyForBoarding(String flightIn, String destination)
  {
    Plane thisPlane = getPlane(flightIn);// throws AirportException if not registered
    if (thisPlane.getStatus().compareTo(PlaneStatus.LANDED)<0)
    {
      throw new AirportException ("flight "+flightIn+" not landed");
    }
    if (thisPlane.getStatus()== PlaneStatus.DEPARTING)
    {
      throw new AirportException ("flight "+flightIn+" already registered to depart");
    }
    thisPlane.upgradeStatus(); // upgrade status from LANDED to DEPARTING
    thisPlane.changeCity(destination); // change city of origin to city of destination
  }
```

The first thing we need to do in this method is to check whether or not an `AirportException` needs to be thrown. The `Javadoc` comments make clear that there are three situations in which we need to throw such an exception.

First, an exception needs to be thrown if the given flight number has not been registered with the airport. At some point we also need to retrieve the `Plane` object from this flight number. Calling the helper method `getPlane` will do both of these things for us, as it throws an `AirportException` if the flight is not registered.

```
// retrieves plane or throws AirportException if flight is not registered
Plane thisPlane = getPlane(flightIn);
```

To check for the remaining exceptions we need to check that the plane currently has the appropriate status to start taking on passengers. The `getStatus` method of a plane returns the status of a plane for us. We know from the previous section that this method returns a value of the enumerated type `PlaneStatus`.

As well as having a `toString` method generated for you when you declare an enumerated type such as `PlaneStatus`, a `compareTo` method (to allow for comparison of two enumerated values) is also generated. This method works in exactly the same way as the `compareTo` method you met when looking at `String` methods. That is, it returns 0 when the two values are equal, a number less than 0 when the first value is less than the second value and a number greater than 0 when the first value is greater than the second value. One enumerated type value is considered *less than* another if it is listed before that value in the original type definition. So, in our example, `DUE` is *less than* `WAITING`, which is *less than* `LANDED` and so on. If a plane has a status that is less than `LANDED` it has not yet landed, so cannot be ready to board passengers—an `AirportException` is thrown:

```
// use 'compareTo' method to compare two status values
if (thisPlane.getStatus().compareTo(PlaneStatus.LANDED)<0)
{
    throw new AirportException ("flight "+flightIn+" not yet landed");
}
```

We also need to throw an `AirportException` if the plane already has a status of `BOARDING`. Although the `compareTo` method can be used to check for equality as well, with most classes it is common to use an `equals` method to do this. An `equals` method is generated for any enumerated type, such as `PlaneStatus`, that you define. However, because of the way enumerated types are implemented in Java, the simple equality operator (`==`) can also be used to check for equality:

```
// equality operator can be used to check if 2 enumerated values are equal
if (thisPlane.getStatus()== PlaneStatus.DEPARTING)
{
    throw new AirportException ("flight "+flightIn+" already registered to depart");
}
```

Having checked for exceptions, we can now indicate that this plane is ready for boarding by upgrading its status (from `LANDED` to `DEPARTING`), and by recording the flights new destination city:

```
// we have cleared all the exceptions so we can update flight details now
thisPlane.upgradeStatus(); // upgrades status from LANDED to DEPARTING
thisPlane.changeCity(destination); // changes city to destination city
```

The inequality operator (`!=`) can be used with enumerated types, to check for inequality of two enumerated type values. An example of this can be seen in the implementation of the `arrivals` method:

```
/**
 * Returns the set of planes due for arrival
 */
public Set<Plane> getArrivals()
{
   Set<Plane> planesOut = new HashSet<Plane>(); // create empty set
   Set<String> items = planes.keySet(); // get all flight numbers
   for(String thisFlight: items) // check status of all
   {
       Plane thisPlane = planes.get(thisFlight);
       if (thisPlane.getStatus() != PlaneStatus.DEPARTING)
       {
          planesOut.add(thisPlane); // add to set
       }
   }
   return planesOut;
}
```

Here we create an empty set of planes. We then add planes into this set if they do not have a status of `DEPARTING`:

```
// use inequality operator to check if status does not equal some value
if (thisPlane.getStatus() != PlaneStatus.DEPARTING)
{
   planesOut.add(thisPlane);
}
```

We have used an enhanced `for` loop in this method, but you might consider using a `forEach` loop. We leave this as end of chapter exercises for you.

Before we leave this section, let us take a look at the `save` and `load` methods that allow us to save and load the attributes in our application. We have three attributes here, `planes` (the `Map` of registered planes), `circlingQ` (the `List` of flight numbers of the planes circling the airport) and `runway` (the array of runways).

Since we have declared our `Plane` and `Runway` classes to be `Serializable`, and because enumerated types such as `PlaneStatus` and collection classes such as `Map` and `List` are already `Serializable`, it is a simple matter to write these objects to a file, and read them from a file. Here is the `save` method:

```
/**
 * Saves airport object to file
 *
 * @param     fileIn The name of the file
 * @throws    IOException if problems with opening and saving to given file
 */
 public void save(String fileIn)throws IOException
 {
     // notice try-with-resources to ensure file closes safely
     try ( FileOutputStream fileOut = new FileOutputStream(fileIn);
           ObjectOutputStream objOut = new ObjectOutputStream (fileOut))
     {
         objOut.writeObject(planes);
         objOut.writeObject(circlingQ);
         objOut.writeObject(runway);
     }
 }
```

You can see we have used a *try-with-resources* construct here to ensure the file is closed once the method terminates.

Here is the `load` method:

```
/**
 * Loads airport object from file
 *
 * @param   fileName The name of the file
 * @throws  IOException if problems with opening and loading given file
 * @throws  ClassNotFoundException if objects in file not of the right type
 */
public void load (String fileName) throws IOException, ClassNotFoundException
{
    // notice try-with-resources to ensure file closes safely
    try ( FileInputStream fileInput = new FileInputStream(fileName);
          ObjectInputStream objInput = new ObjectInputStream (fileInput))
    {
        planes = (Map<String, Plane>) objInput.readObject();
        circlingQ = (List<String>) objInput.readObject();
        runway = (Runway[])objInput.readObject();
    }
}
```

Again, we have used a *try-with-resources* construct to ensure our file is closed upon termination. Notice that when we load the attributes from file, we must indicate their type. The collection class types need to be marked using the generics mechanism:

```
// indicate the type of each collection using generics mechanism
planes = (Map<String, Plane>) objInput.readObject();
circlingQ = (List<String>) objInput.readObject();
runway = (Runway[])objInput.readObject();
```

There is nothing particularly new in the remaining methods. Take a look at the comments provided to follow their implementation.

21.7 Testing

In Chaps. 11 and 12 we looked at the concepts of unit testing and integration testing. We have left unit testing to you as a practical task, but we will spend a little time here considering integration testing. A useful technique to devise test cases during integration testing is to review the behaviour specifications of use cases, derived during requirements analysis.

Remember, a use case describes some useful service that the system performs. The behaviour specifications capture this service from the point of view of the user. When testing the system you take the place of the user, and you should ensure that the behaviour specification is observed.

Often, there are several routes through a single use case. For example, when registering a plane, either the plane could be successfully registered, or an error is indicated. Different routes through a single use case are known as different

scenarios. During integration you should take the place of the user and make sure that you test *each* scenario for *each* use case. Not surprisingly, this is often known as **scenario testing**. As an example, reconsider the "*Record flight's request to land*" use case:

An air traffic controller records an incoming flight entering airport airspace, and requesting to land at the airport, by submitting its flight number. As long as the plane has previously registered with the airport, the air traffic controller is given an unoccupied runway number on which the plane will have permission to land. If all runways are occupied however, this permission is denied and the air traffic controller is informed to instruct the plane to circle the airport. If the plane has not previously registered with the airport an error is signalled.

From this description three scenarios can be identified:

Scenario 1
An air traffic controller records an incoming plane entering airport airspace and requesting to land at the airport, by submitting its flight number, and is given an unoccupied runway number on which the plane will have permission to land.

Scenario 2
An air traffic controller records an incoming plane entering airport airspace and requesting to land at the airport, by submitting its flight number. The air traffic controller is informed to instruct the plane to circle the airport as all runways are occupied.

Scenario 3
An air traffic controller records an incoming plane entering airport airspace and requesting to land at the airport, by submitting its flight number. An error is signalled as the plane has not previously registered with the airport.

Similar scenarios can be extracted for each use case. During testing we should walk through each scenario, checking whether the outcomes are as expected.

21.8 Design of the JavaFX Interface

Figure 21.4 illustrates the interface design we have chosen for the *Airport* application. A few new JavaFX features have been highlighted.

Apart from the features highlighted in Fig. 21.4, the remaining JavaFX components will be familiar to you. The three new JavaFX features are: a layout component known as a **tabbed pane**; some text that appears when you keep your cursor over a component—known as that component's **tool tip** and a `Stage` with

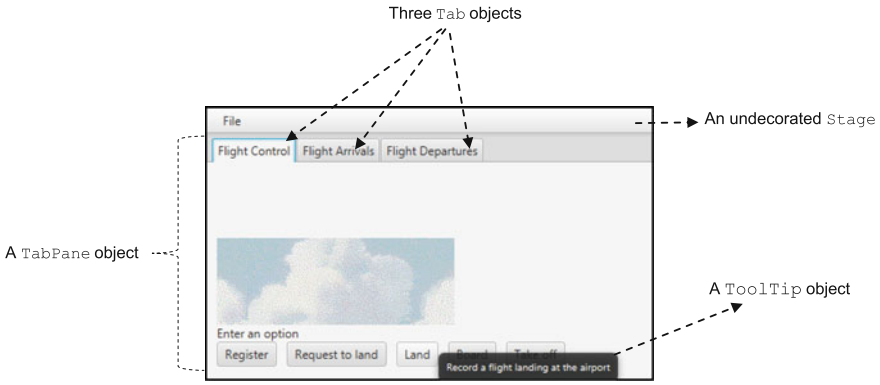

Fig. 21.4 Some new JavaFX features in the *Airport* JavaFX interface

no icons for minimising/maximising/closing—known as an **undecorated** stage. We will return to look at tool tips and undecorated stages later in the chapter but we will look at a tabbed pane now—it is implemented in JavaFX using a `TabPane` class.

21.9 The *TabPane* Class

The `TabPane` class provides a very useful JavaFX component for organizing the user interface. You can think of a `TabPane` component as a collection of overlapping tabbed "cards", on which you place other user interface components. A particular card is revealed by clicking on its **tab**. This allows certain parts of the interface to be kept hidden until required, thus reducing screen clutter.

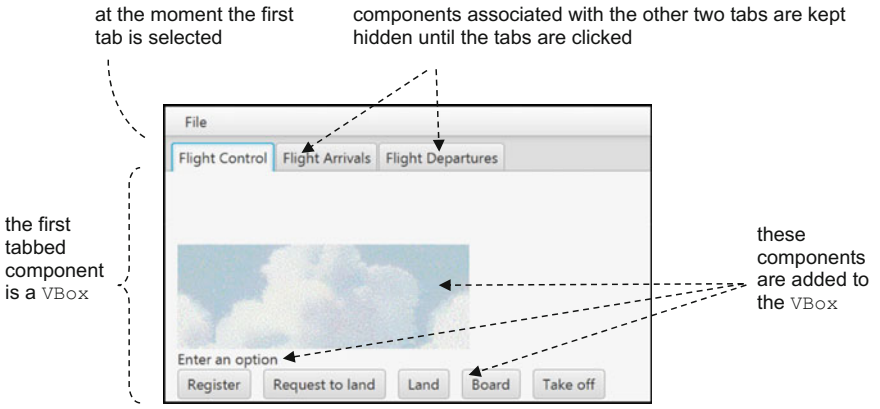

Fig. 21.5 A `TabPane` allows parts of the interface to be revealed selectively

A `TabPane` component can consist of any number of tabbed cards. Each card is actually a *single* component of your choice. If you use a container component such as a `VBox`, you can effectively associate many components with a single tab (see Fig. 21.5).

We can construct a `TabPane` component by calling the empty constructor as follows:

```
TabPane tabbedPane = new TabPane();
```

A `TabPane` can be associated with any number of individual tabbed cards. To do this we use the `Tab` class for each individual tabbed card. We need three tabbed cards, one for the main screen shown in Fig. 21.5, one for the arrivals information and one for the departures information:

```
Tab tab1 = new Tab("Flight Control"); // main flight control tab
Tab tab2 = new Tab("Flight Arrivals"); // arrivals tab
Tab tab3 = new Tab("Flight Departures"); // departures tab
```

The strings provided to the constructors are the titles displayed on each tab.

We can now add tabbed components to the `TabPane`. When adding a tabbed component to a `TabPane`, you call the `getTabs` method to access the link to the collection of tabbed cards and then the tabs themselves can be set using the `addAll` method:

```
tabbedPane.getTabs().addAll(tab1, tab2, tab3);
```

The order in which the tabs are given to the `addAll` method is the order in which they will appear on the screen.[1] As we mentioned, each tabbed card is associated with a single component. So to add multiple components to a tab we can use a container such as a `VBox` or a `HBox`. For example, to create the main tab shown in Fig. 21.5, we have 3 components highlighted—a cloud image (named `imageView` in the code), a label (named `label` in the code) and a collection of buttons (stored in a `HBox` named `controls` in the code). We add all three of these to a `VBox` (named `box` in the code):

```
// add cloud image, label and button collection to VBox
box.getChildren().addAll(imageView,label, controls);
```

[1]By default, the tabs you add will appear at the top left of the `TabPane` (as in Fig. 21.4). A `setSide` method can be used to choose an alternative side (the top right, the bottom, the left or the right).

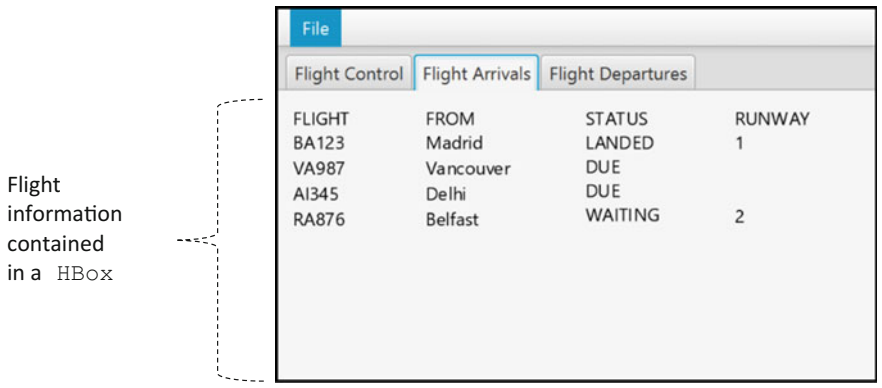

Fig. 21.6 Both "Flight Arrivals" and "Flight Departures" tabs consist of flight information in a HBox

We then use the setContent method of a tabbed card to add this VBox component to a given tab:

```
// add VBox to tab1
tab1.setContent(box);
```

The "Flight Arrivals" and "Flight Departures" each contain a HBoxes (each of which contains a series of VBox columns) to display arrivals and departure information respectively. Figure 21.6 shows the airport GUI after selecting the "Flight Arrivals" tab.

21.10 The *AirportFrame* Class

We now present the complete code for the JavaFX class. Take a look at it and then we will point out a few features:

AirportFrame

```
package airportSys; // add class to package

import java.util.Set;

import javafx.application.Application;
import static javafx.application.Application.launch;
import javafx.application.Platform;
import javafx.geometry.Insets;
```

```
import javafx.geometry.Pos;
import javafx.scene.Scene;
 mport javafx.scene.control.Alert;
import javafx.scene.control.Alert.AlertType;
import javafx.scene.control.Button;
import javafx.scene.control.ButtonType;
import javafx.scene.control.Label;
import javafx.scene.control.Menu;
import javafx.scene.control.MenuBar;
import javafx.scene.control.Tab;
import javafx.scene.control.TabPane;
import javafx.scene.control.TextInputDialog;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import javafx.scene.layout.HBox;
import javafx.stage.Stage;
import javafx.scene.layout.VBox;
import javafx.scene.text.Text;
import javafx.scene.control.Tooltip;
import javafx.scene.layout.Border;
import javafx.scene.layout.BorderStroke;
import javafx.scene.layout.BorderStrokeStyle;
import javafx.scene.layout.BorderWidths;
import javafx.scene.layout.CornerRadii;
import javafx.scene.paint.Color;
import javafx.stage.StageStyle;

/**
 * Class to provide the JavaFX interface of the airport system
 *
 * @author Charatan and Kans
 * @version 6th August 2018
 */

public class AirportFrame extends Application
{
    // declare Airport object
    private Airport myAirport;
    // additional data required for the airport system
    private int numberOfRunways ;
    private final String FILENAME = "airport.dat";

    // create arrival and departure visual components that need global access

    // arrivals information displayed in a HBox
    private HBox arrivals = new HBox(50);
    // include columns for arrivals information
    private   VBox arrivalsColumn1 = new VBox();
    private   VBox arrivalsColumn2 = new VBox();
    private   VBox arrivalsColumn3 = new VBox();
    private   VBox arrivalsColumn4 = new VBox();
    // departures information displayed in a HBox
    private   HBox departures = new HBox(60);
    // include columns for departures information
    private   VBox departuresColumn1 = new VBox();
    private   VBox departuresColumn2 = new VBox();
    private   VBox departuresColumn3 = new VBox();

    // methods

    /**
     * The start method to initialise the screen and the airport data
     *
     * @param stage The Stage object
     */
    @Override
    public void start(Stage stage)
    {
        // check if data is to be loaded from file
        Alert alert = new Alert(  AlertType.INFORMATION, "Do you want to restore your data?",
                                  ButtonType.YES, ButtonType.NO);
        String response = alert.showAndWait().get().getText();
        if (response.equals("Yes")) // load data from file
        {
            try
            {
                myAirport = new Airport(FILENAME); // call file loading constructor
                listArrivals(); // update arrivals tab
                listDepartures(); // update departures tab
                showInfo("Planes loaded");
            }
            catch (Exception e) // file loading errors
            {
                showError("File Opening error");
                System.exit(1); // indicates exit with error
            }
```

```
        }
        else // initialise an empty airport
        {
            numberOfRunways = getNumberOfRunways(); // request number of runways
            try
            {
                myAirport = new Airport (numberOfRunways); // create an empty Airport object
            }
            catch (AirportException ae) // error initialising Airport object
            {
                showError(ae.getMessage());
                System.exit(1); // indicates exit with error
            }
            catch (Exception e) // in case of any unforseen error
            {
                showError(e.getMessage());
                System.exit(1); // indicates exit with error
            }
        }
        // set up three Tab objects in a TabPane
        TabPane tabbedPane = new TabPane();
        Tab tab1 = new Tab("Flight Control"); // main flight control tabs
        Tab tab2 = new Tab("Flight Arrivals"); // arrivals tab
        Tab tab3 = new Tab("Flight Departures"); // departures tab
        tabbedPane.getTabs().addAll(tab1, tab2, tab3);
        // ensure tabs remain open
        tab1.setClosable(false);
        tab2.setClosable(false);
        tab2.setClosable(false);

        // creat a VBox to hold all scene components
        VBox root = new VBox();

        // set up menu bar and items
        MenuBar bar = new MenuBar();
        bar.setMinHeight(25);
        Menu item = new Menu("File");
        Menu saveAndContinueOption = new Menu("Back-up and continue");
        Menu saveAndExitOption = new Menu("Back-up and exit");
        Menu exitWithoutSavingOption = new Menu("Exit without backing-up");
        item.getItems().addAll(saveAndContinueOption, saveAndExitOption, exitWithoutSavingOption);
        bar.getMenus().add(item);

        // create and customise a VBox to organise flight control screen
        VBox box = new VBox();
        box.setPadding(new Insets(10));
        box.setMinHeight(215);
        // add Vbox to tab1
        tab1.setContent(box);
        box.setAlignment(Pos.BOTTOM_LEFT);
        // create a cloud image
        Image image = new Image("clouds.png");
        ImageView imageView = new ImageView(image);
        // create an instructional label
        Label label = new Label("Enter an option");
        // create a HBox to hold main flight control buttons
        HBox controls = new HBox(10);
        // create flight control buttons and add tooltips
        Button button1 = new Button("Register");
        button1.setTooltip(new Tooltip("Register a flight with the airport"));
        Button button2 = new Button("Request to land");
        button2.setTooltip(new Tooltip("Record a flight requesting to land at the airport"));
        Button button3 = new Button("Land");
        button3.setTooltip(new Tooltip("Record a flight landing at the airport"));
        Button button4 = new Button("Board");
        button4.setTooltip(new Tooltip("Record a landed flight ready for boarding new passengers"));
        Button button5 = new Button("Take off");
        button5.setTooltip(new Tooltip("Record a flight leaving the airport"));
        // add buttons to HBox
        controls.getChildren().addAll(button1, button2, button3, button4, button5);
        // add cloud image, label and button collection to VBox
        box.getChildren().addAll(imageView,label, controls);
        try
        {
            // code button responses by calling private methods
            button1.setOnAction(e -> register());
            button2.setOnAction(e -> requestToLand());
            button3.setOnAction(e -> land());
            button4.setOnAction(e -> board());
            button5.setOnAction(e -> takeOff());
            // code menu responses
            saveAndContinueOption.setOnAction(e -> save(FILENAME));
            saveAndExitOption.setOnAction(e ->{
                                              save(FILENAME);
                                              Platform.exit();
                                            });
            exitWithoutSavingOption.setOnAction(e -> exitWithoutSaving());
```

```
        }
        catch(Exception e) // for any unforseen errors
        {
            showError("Invalid Operation");
        }

        // customise look of arrivals tab
        arrivals.setPadding(new Insets(10));
        arrivals.getChildren().addAll(     arrivalsColumn1, arrivalsColumn2, arrivalsColumn3,
                                           arrivalsColumn4);
        tab2.setContent(arrivals);
        // customise look of departures tab
        departures.setPadding(new Insets(10));
        departures.getChildren().addAll(departuresColumn1, departuresColumn2, departuresColumn3);
        tab3.setContent(departures);
        // customise root object and add menu and tabbed pane
        root.setBorder( new Border(new BorderStroke(Color.BLACK, BorderStrokeStyle.SOLID,
                        new CornerRadii(0), new BorderWidths(2))));
        root.getChildren().addAll(bar, tabbedPane);
        // customise frame
        Scene scene = new Scene(root,450, 275);
        stage.setScene(scene);
        stage.setTitle("Airport System");
        stage.initStyle(StageStyle.UNDECORATED); // for undecorated frame

        stage.show();
    }

    /**
     *  Private method to request and return the number of runways
     */
    private int getNumberOfRunways()
    {
        TextInputDialog dialog = new TextInputDialog();
        dialog.setHeaderText("Enter number of runways");
        dialog.setTitle("Runway Information Request");

        String response = dialog.showAndWait().get();

        if (!response.equals("")) // check for empty string
        {
            return Integer.parseInt(response);
        }
        else
        {
            return -1; // to indicate no runway set
        }
    }

    /**
     *  Private method to register new flight with the airport
     */
    private void register()
    {
        String flightNo, city;
        try
        {
            TextInputDialog dialog = new TextInputDialog();
            dialog.setHeaderText("Enter flight number");
            dialog.setTitle("Registration form");
            flightNo = dialog.showAndWait().get();

            // throws AirportException if no flight entered
            checkIfEmpty(flightNo, "No flight number entered");

            dialog = new TextInputDialog();
            dialog.setHeaderText("Enter city of origin");
            dialog.setTitle("Registration form");
            city = dialog.showAndWait().get();

            // throws AirportException if no city entered
            checkIfEmpty(city, "No city entered");

            // register flight
            myAirport.registerFlight(flightNo, city);
            showInfo("confirmed:\nflight "+flightNo +" registered from "+city);
        }
        catch(AirportException ae) // catch airport exceptions
        {
            showError(ae.getMessage());
        }
        listArrivals();  // update arrivals tab
    }

    /**
```

```
     *  Private method to record a flight's request to land
     */
    private void requestToLand()
    {
        String flightNo, message;

        try
        {
            TextInputDialog dialog = new TextInputDialog();
            dialog.setHeaderText("Enter flight number");
            dialog.setTitle("Request to land form");
            flightNo = dialog.showAndWait().get();

            // throws AirportException if no flight entered
            checkIfEmpty(flightNo, "No flight number entered");

            // record flight's request to land and get runway number
            int runway = myAirport.arriveAtAirport(flightNo);
            // check runway number
            if (runway == 0)
            {
                message = "no runway available, circle the airport";
            }
            else
            {
               message = " land on runway "+runway;
            }
            showInfo("confirmed:\nflight "+flightNo + message);
        }
        catch(AirportException ae) // catch airport exceptions
        {
            showError(ae.getMessage());
        }
        listArrivals(); // update arrivals tab
    }

    /**
     *  Private method to record a flight landing at the airport
     */
    private void land()
    {
        String flightNo, runwayIn;
        int runway;

        try
        {
            TextInputDialog dialog = new TextInputDialog();
            dialog.setHeaderText("Enter flight number");
            dialog.setTitle("Landing form");
            flightNo = dialog.showAndWait().get();

            // throws AirportException if no flight entered
            checkIfEmpty(flightNo, "No flight number entered");

            dialog = new TextInputDialog();
            dialog.setHeaderText("Enter runway number");
            dialog.setTitle("Landing form");
            runwayIn = dialog.showAndWait().get();

            // throws AirportException if no runway enetered
            checkIfEmpty(flightNo, "No flight number entered");

            // convert runway to an integer
            runway = Integer.parseInt(runwayIn);
            // record flight landing
            myAirport.landAtAirport(flightNo, runway);
            showInfo("confirmed:\nflight "+flightNo +" landed on runway "+runway);
        }
        catch (AirportException ae) // catch airport exceptions
        {
           showError(ae.getMessage());
        }
        listArrivals(); // update arrivals tab
    }

    /**
     *  Private method to register a flight boarding passengers at the airport
     */
    private void board ()
    {
        String flightNo, city;

        try
        {
            TextInputDialog dialog = new TextInputDialog();
            dialog.setHeaderText("Fight number");
            dialog.setTitle("Boarding form");
```

```
            flightNo = dialog.showAndWait().get();

            // throws AirportException if no flight entered
            checkIfEmpty(flightNo, "No flight number entered");

            dialog = new TextInputDialog();
            dialog.setHeaderText("Enter destination city");
            dialog.setTitle("Boarding form");
            city = dialog.showAndWait().get();

            // throws AirportException if no city entered
            checkIfEmpty(city, "No city entered");

            // record flight boarding
            myAirport.readyForBoarding(flightNo, city);
            showInfo("confirmation:\nflight "+flightNo+" boarding to "+city);
        }
        catch (AirportException ae) // catch airport exceptions
        {
           showError(ae.getMessage());
        }
        listArrivals(); // update arrivals tab
        listDepartures(); // update departures tab
    }

    /**
     *  Private method to register a flight leaving the airport
     */
    private void takeOff ()
    {
        String flightNo;

        try
        {
            TextInputDialog dialog = new TextInputDialog();
            dialog.setHeaderText("Fight number");
            dialog.setTitle("Take off form");
            flightNo = dialog.showAndWait().get();

            // throws AirportException if no flight entered
            checkIfEmpty(flightNo, "No flight number entered");

            // record flight taking off
            myAirport.takeOff(flightNo);
            showInfo("confirmation:\nflight "+flightNo+" Removed from system");
        }
        catch (AirportException ae) // catch airport exceptions
        {
           showError(ae.getMessage());
        }
        listDepartures(); // update departures tab
    }

    /**
     *  Private method to update arrivals tab information
     */
    private void listArrivals()
    {
      // get arrivals information
      Set<Plane> arrivalsList = myAirport.getArrivals();
      // clear current arrivals information
      arrivalsColumn1.getChildren().clear();
      arrivalsColumn2.getChildren().clear();
      arrivalsColumn3.getChildren().clear();
      arrivalsColumn4.getChildren().clear();
      arrivalsColumn1.getChildren().add(new Text("FLIGHT"));
      arrivalsColumn2.getChildren().add(new Text("FROM"));
      arrivalsColumn3.getChildren().add(new Text("STATUS"));
      arrivalsColumn4.getChildren().add(new Text("RUNWAY"));
      // re-populate arrivals information
      for (Plane thisPlane: arrivalsList)
      {
        arrivalsColumn1.getChildren().add(new Text(thisPlane.getFlightNumber()));
        arrivalsColumn2.getChildren().add(new Text(thisPlane.getCity()));

        arrivalsColumn3.getChildren().add(new Text(thisPlane.getStatusName()));
        try
        {
            // throws exception if no runway set
            arrivalsColumn4.getChildren().add(
                                        new Text(Integer.toString(thisPlane.getRunwayNumber())));
        }
        catch(Exception e) // catch exception and leave runway column blank
        {
             arrivalsColumn4.getChildren().add(new Text(""));
        }
     }
```

```
    }

    /**
     *  Private method to update departures tab information
     */
    private void listDepartures()
    {
      // get departures information
      Set<Plane> departuresList = myAirport.getDepartures();
      // clear current departures information
      departuresColumn1.getChildren().clear();
      departuresColumn2.getChildren().clear();
      departuresColumn3.getChildren().clear();
      departuresColumn1.getChildren().add(new Text("FLIGHT"));
      departuresColumn2.getChildren().add(new Text("TO"));
      departuresColumn3.getChildren().add(new Text("RUNWAY"));
      // re-populate departures information
      for (Plane thisPlane: departuresList)
      {
        departuresColumn1.getChildren().add(new Text(thisPlane.getFlightNumber()));
        departuresColumn2.getChildren().add(new Text(thisPlane.getCity()));

        try
        {
            // throws exception if no runway set
            departuresColumn3.getChildren().add(
                                        new Text(Integer.toString(thisPlane.getRunwayNumber())));
        }
        catch(Exception e) // catch exception and leave runway column blank
        {
            departuresColumn3.getChildren().add(new Text(""));
        }
     }
    }

    /**
     *  Private method to exit application without saving data
     */
    private void exitWithoutSaving()
    {
        Alert alert = new Alert(  AlertType.WARNING, "Are you sure? Your work could be lost.",
                                  ButtonType.YES, ButtonType.CANCEL);
        alert.setTitle("Confirmation reqired");
        String response = alert.showAndWait().get().getText();

        if(response.equals("Yes"))
        {
            Platform.exit();
        }
    }

    /**
     *  Private method to load airport data from a file
     *  @param   fileName The name of the file to open
     */
    private void open(String fileName)
    {
        try
        {
            myAirport.load(fileName); // may throw an exception
            listArrivals(); // update arrivals tab
            listDepartures(); // update departures tab
            showInfo("Planes Loaded");
        }
        catch (Exception e) // catch file related exceptions
        {
            showError("File Opening error");
            System.exit(1); // indicates exit with error
        }
    }

    /**
     *  Private method to save airport data to a file
     *  @param   fileName The name of the file to save
     */
         private void save(String fileName)
    {
        try
        {
            myAirport.save(fileName); // may throw exception
            showInfo("Planes saved");
        }
        catch (Exception e) // catch file related exceptions
        {
            showError("Error saving data");
        }
```

```
    }

    /**
     *  Private method to show an error message
     *  @param   msg The error message
     */
    private void showError(String msg)
    {
        Alert alert = new Alert(AlertType.ERROR);
        alert.setHeaderText("Airport Error Alert");
        alert.setContentText(msg);
        alert.showAndWait();
    }

    /**
     *  Private method to show an information message
     *  @param   msg The information message
     */
    private void showInfo(String msg)
    {
        Alert alert = new Alert(AlertType.INFORMATION);
        alert.setHeaderText("Airport Information Alert");
        alert.setContentText(msg);
        alert.showAndWait();
    }

    /**
     *  Private method to check if a string is empty
     *  @param  s The string to check
     *  @param  errorMessage The error message to include in an exception
     *  @throws AirportException if string is empty
     */
    private void checkIfEmpty(String s, String errorMsg)
    {
        if (s.equals(""))
        {
            throw new AirportException (errorMsg);
        }
    }

    public static void main(String[] args)
    {
        launch(args);
    }
}
```

As you can see, although this class has more code than those we have met before, it follows a familiar pattern. Most of the code will therefore be familiar to you and the `Javadoc` comments and additional supplementary comments should be sufficient to follow what we have done here. We will just draw your attention to one or two JavaFX features that we have decided to incorporate into our implementation that will be new to you and that we mentioned in the introduction.

First, we have added **tool tips** to our buttons. A tool tip is an informative description of the purpose of a GUI component. This informative description is revealed when the user places the cursor over the component. Figure 21.7 shows the tool tip that is revealed when the cursor is placed over the "Land" button.

Adding a tool tip to a JavaFX component is easy; just use the `setToolTipText` method and provide a tool tip message as a parameter to the `ToolTip` class:

Fig. 21.7 A tool tip is revealed when the mouse is placed over the "Land" button

```
// create a Land button
Button button3 = new Button("Land");
// add a tool tip to the Land button using the setToolTip method
button3.setTooltip(new Tooltip("Record a flight landing at the airport"));
```

We also mentioned that we have made our frame an undecorated frame. An undecorated frame has no icons to minimise or maximise (or close) the frame. This ensures the frame remains the same size and this cannot be altered. To do this we use the `initStyle` method of our `stage` object and pass an enumerated type constant `StageStyle.UNDECORATED` as a parameter:

```
stage.initStyle(StageStyle.UNDECORATED);
```

Finally we draw your attention to a few things related to the functionality of this class.

When the application opens the user is given the option to load data from a file via an `Alert` dialog (see Fig. 21.8).

If the user chooses to load data from a file, the airport interface (as given in Fig. 21.5) is activated once the data is loaded. If the user chooses not to load data from a file, the system needs to be initialised by asking the user for the number of runways associated with this airport. A `TextInputDialog` is used for this purpose (see Fig. 21.9).

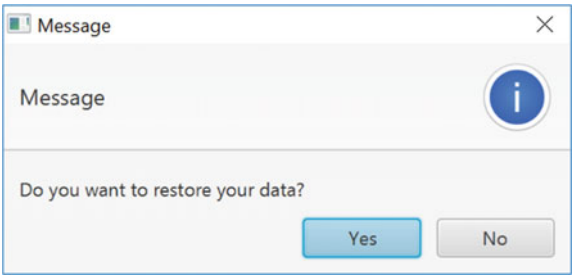

Fig. 21.8 An `Alert` dialog to allow users to load data from a file

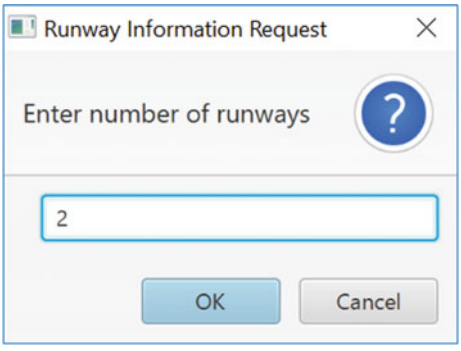

Fig. 21.9 A `TextInputDialog` to allow users to specify the number of runways

Again, the airport interface (as given in Fig. 21.5) is activated once the user has entered the number of runways. The user can then use the buttons on the flight control tab to register, request to land, land, board and take off flights at the airport —with arrivals and departures information being always available in the arrivals and departures tabs.

We have created two **`private`** methods, `listArrivals` and `listDepartures`. These methods update the information in the arrivals and departures tabs respectively. Whenever the airport data is modified (i.e. when a button is pressed in the main control tab) these methods need to be called to update the arrivals and departures tabs so when these tabs are revealed they always show the current state of flights in the system.

For example, when a flight registers at the airport the arrivals tab only needs to be updated before we exit the method. Here is the outline of the code for processing the register button response:

```
/**
 *  Private method to register new flight with the airport
 */
private void register()
{
       // code to register flight at airport here

       listArrivals();  // update arrivals tab
}
```

However, when a flight is ready for boarding, it needs to be removed from the arrivals tab and added to the departures tab, so both `listArrivals` and `listDepartures` need to be called.

```
/**
 *  Private method to register a flight boarding passengers at the airport
 */
private void board()
{
       // code to record flight as ready for boarding

       listArrivals();  // update arrivals tab
       listDepartures();  // update departures tab
}
```

When a flight takes off from the airport, it only needs to be removed from the departures tab:

```
/**
 *  Private method to record a flight leaving the airport
 */
private void takeOff()
{
       // code to record flight leaving the airport

       listDepartures();  // update departures tab only
}
```

Finally, the user can use the file menu to back-up the data to a file and continue with the application, to back-up data to a file and exit the application or simply exit the application without backing up the data (see Fig. 21.10).

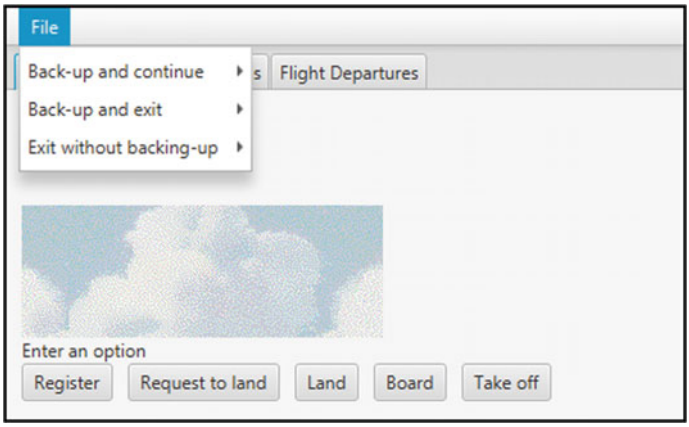

Fig. 21.10 Menu options to allow users to back-up data and exit the application as required

21.11 Self-test Questions

1. In Sect. 21.6 we developed scenarios for the use case "*Register flight arrival*". Develop scenarios for all the other use cases in Table 21.1.
2. What is the difference between *containment* and *composition* in UML?
3. Consider an enumerated type, `Light`. This type can have one of three values: RED, AMBER and GREEN. It will be used to display a message to students, indicating whether or not a lecturer is available to be seen.

   (a) Specify this type in UML.
   (b) Implement this type in Java.
   (c) Declare a `Light` variable, `doorLight`;
   (d) Write a **`switch`** statement that checks `doorLight` and displays "I am away" when `doorLight` is RED, "I am busy" when `doorLight` is AMBER and "I am free" when `doorLight` is GREEN.

4. Identify the benefits offered by the `TabPane` component.
5. How can the tool tip "*This button stops the game*" be added to a `Button` called `stop`?
6. Develop test plans for the `Runway`, `Plane`, `Airport` and `AirportFrame` classes.

21.12 Programming Exercises

Copy, from the accompanying web-site, the classes that make up the airport application and then tackle the following exercises:

1. Develop `toString` methods for the `Runway`, `Plane` and `Airport` classes and then develop testers for these classes.
2. Run and test the classes in the airport application by making use of your testing programs of programming exercise 1 above and following your test plans you devised in self-test question 6.
3. Make any further enhancements that you wish to the airport application. For example, you may wish to consider

   (a) adding a fourth "Help" tab that displays text describing how to use the airport application;
   (b) rather than use an enhanced **`for`** loop in the `getArrivals` and `getDepartures` methods of the `Airport` class make use of **`forEach`** loops;
   (c) instead of a `TextInputDialog` to enter the runway number in the `land` method of the `AirportFrame` class, use a `ChoiceDialog` with a list of runways pre-populated in a drop-down box;
   (d) identify methods, such as `nextFreeRunway` and `nextToLand` in the `Airport` class, that could return **`null`** values and then modify the code in the airport application to make use of the `Optional` type to avoid returning these **`null`** values;
   (e) design your own skin for the `AirportFrame` by creating a cascading style sheet.

4. Create an executable JAR file to run the airport application.