Manchester Metropolitan University

Department of Computing and Mathematics

ASSIGNMENT COVER SHEET

Unit title:	Advanced Programming
Assignment set by:	K. Welsh
Assignment ID:	1CWK50
Assignment title:	Airline Scheduling System
Assessment weighting:	50%
Type: (Group/Individual)	Individual
Hand-in deadline:	**21:00 10th January 2020**
Hand-in format and mechanism:	Online, via Moodle

Learning outcomes being assessed:

LO1: Design and implement object oriented applications and understand principles relating to interactive user interface development.
LO2: Use tools and techniques and utilise existing classes and libraries in a systematic way to develop a complex software application.
LO5: Analyse and produce abstract designs to produce robust software designs.

Note: it is your responsibility to make sure that your work is complete and available for marking by the deadline. Make sure that you have followed the submission instructions carefully, and your work is submitted in the correct format, using the correct hand-in mechanism (e.g. Moodle upload). If submitting via Moodle, you are advised to check your work after upload, to make sure it has uploaded properly. Do not alter your work after the deadline. You should make at least one full backup copy of your work.

Penalties for late hand-in: see Regulations for Undergraduate Programmes of Study (http://www.mmu.ac.uk/academic/casqe/regulations/assessment.php). The timeliness of submissions is strictly monitored and enforced.

All coursework has a late submission window of 5 working days, but any work submitted within the late window will be capped at 40%, unless you have an agreed extension. Work submitted after the 5-day window will be capped at zero, unless you have an agreed extension.

Please note that individual tutors are unable to grant extensions to coursework.

Exceptional Factors affecting your performance: see Regulations for Undergraduate Programmes of Study (http://www.mmu.ac.uk/academic/casqe/regulations/assessment/docs/ug-regs.pdf). For advice relating to exceptional factors, please see the following website: https://www2.mmu.ac.uk/student-case-management/guidance-for-students/exceptional-factors/ or visit a Student Hub for more information.

Plagiarism: Plagiarism is the unacknowledged representation of another person's work, or use of their ideas, as one's own. Manchester Metropolitan University takes care to detect plagiarism, employs plagiarism detection software, and imposes severe penalties, as outlined in the Student Handbook (http://www.mmu.ac.uk/academic/casqe/regulations/docs/policies_regulations.pdf and Regulations for Undergraduate Programmes (http://www.mmu.ac.uk/academic/casqe/regulations/assessment.php). Bad referencing or submitting the wrong assignment may still be treated as plagiarism. If in doubt, seek advice from your tutor.

As part of a plagiarism check, you may be asked to attend a meeting with the Unit Leader, or another member of the unit delivery team, where you will be asked to explain your work (e.g. explain the code in a programming assignment). If you are called to one of these meetings, it is very important that you attend.

Assessment Criteria:	Indicated in the attached assignment specification.
Formative Feedback:	Formative feedback is available from lab tutors in a dedicated lab session towards the end of the Autumn term, as well as via a formative submission on the 13th December 2019
Summative Feedback Format:	Feedback will be generated by an automated tester, returned via Moodle, along with the assignment grade

Advanced Programming

Assignment 1: Airline Scheduling & Rostering System

Deadline: 21:00 10th January 2020

Introduction

In this assignment, you will be creating some of the key components of a system that can plan which plane an airline uses to fly passengers on each of their flights, and choose which pilots and cabin crew are rostered for each flight. This is a simplified but representative version of a real problem faced by airlines and other businesses. The University has consulted with several businesses in other sectors in the region to develop this kind of software over recent years.

Because large aircraft weigh more and consume more fuel than smaller ones, airlines try to allocate the smallest aircraft with enough seats to accommodate the booked passengers to a flight. Pilots and cabin crew, however, are only qualified to operate certain types of aircraft. Such "type ratings" can take a considerable amount of time and money to obtain, so it is rare for staff to be qualified to operate all the types of aircraft that an airline may possess.

The rules governing how long and how often airline staff may work are complex and strict. They are designed to ensure passenger safety, and are negotiated nationally and internationally between pilot and cabin crew unions and government safety agencies. Thus, deciding which type of plane to fly on a particular route on a particular day is a complex decision factoring in passenger demand, the availability of aircraft, and the availability of qualified pilots and cabin crew.

This assignment simulates some of the constraints you could find yourself working under when developing software in industry. You will be supplied with certain key interfaces to which your software must conform, and your system will have to operate on data supplied in various formats, as if it were exported from separate existing, so-called *legacy systems*. You will also have to use specific libraries to complete the work, although these are the same ones you used in the relevant lab sessions.

Getting Started

To start work on the assignment, you will need to import the starter project, available to download on Moodle, into Eclipse. To perform the import, you will need to select file→import from the menu in eclipse, then select General, then "Existing Projects into Workspace". This opens the "Import Projects" dialog, as depicted below.

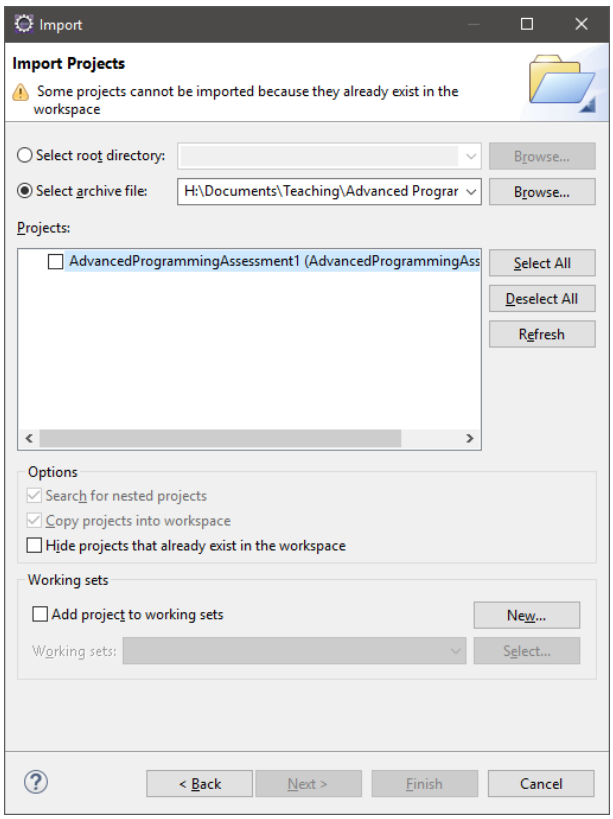

Figure 1: Eclipse Import Projects Dialog

In this dialog, you will need to select the "Select archive file" radio button, navigate to the starter project zip file that you downloaded from Moodle, and then tick the checkbox next to the "AdvancedProgrammingAssessment1" project on the list below. Click "Finish" to complete the import process. The starter project should appear in Eclipse's Package Explorer on the left of your screen.

Project Structure

The starter project comes with quite a few files organised into various folders. Its worth spending a moment to find your way around the project. A screenshot of the "Package Explorer" view in Eclipse of the project is depicted below, for reference:

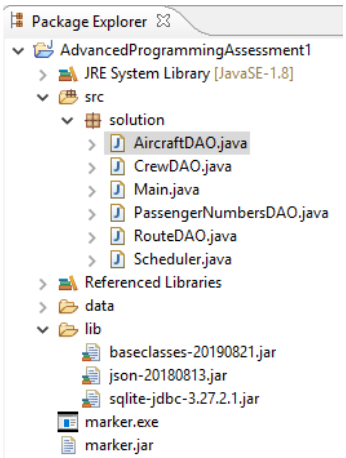

Figure 2: Package Explorer
View of Starter Project

The **src** folder of the project contains all of the project's source code, as you've seen previously. You can see in the screenshot above that all of the classes form part of the "solution" package. Don't change this or add java classes outside of this package: the automatic feedback generator will not be able to find them. Each of the java classes corresponds to a section of the assignment, implementing the correct interface and containing the correct method signatures. The Main class contains a simple program to help you test the loading of the aircraft data, for the first section of the assignment.

The **Referenced Libraries** depicted above isn't actually a folder. It just lists the various jar files upon which this project depends. The entries in here match those in the **lib** folder further down the list. The project has three dependencies: the xerial SQLite JDBC connector and the JSON reference parser, both of which you've used previously in labs, and the "baseclasses" jar file. The base classes are things like the Aircraft, Crew, and Schedule classes, which are ready made for you and you will not be able to modify.

The **data** folder contains a number of data files that you'll be writing java classes responsible for loading and searching through, as well as a number of files used internally by the automatic feedback generator. If you modify any of the files in the data folder, the changes will be lost next time you run the automatic feedback generator: it re-creates the correct versions of the files each time you run it.

Finally, the **marker.exe** and **marker.jar** files are the automatic feedback generator. On most Windows systems, you should be able to double click the marker.exe file to run the generator directly. On other systems, you may need to navigate to the project folder on the command line and run the program using the following command:

```
java -jar marker.jar
```

However you run the automatic feedback generator, when it launches it will run a series of tests on the code that you have written, save the results into a file named feedback.pdf, which it will attempt to open for viewing. Make sure that you close the PDF before running the generator again, as the process will fail if it can't open the file for writing!

The output produced by the automatic feedback generator tells you which tests your work is currently passing, which it is currently failing and, crucially, offers a brief explanation as to why your work is failing the test. Normal unit tests do not offer any explanation of why code fails a particular test, but you are still learning!

JavaDoc

Also available on Moodle is a complete copy of the HTML documentation for each of the classes in the base classes jar file. This will help you to use the functionality provided to you in the base classes. It will also help you to see what methods each class has, and what the parameters and return types of the methods you will need to implement are.

The System

You are going to create various parts of a system that, if completed, would be capable of allocating aircraft and staff to routes on particular dates, factoring in passenger demand and the various rules governing the working pattern of crew. For this assignment, you do not need to worry about the user interface. Instead, you'll be focussing on the classes responsible for loading data into the system and, later on, developing your own scheduling algorithm.

The assignment is made up of a number of sections, each of which is focussed on a specific area of the system. The following sections discuss each part of the assignment in turn. It is recommended that you complete your work in the same order as the sections, as some of the later parts become increasingly difficult.

The Aircraft Data (20 marks)

The system that holds data on the airline's aircraft is the oldest of the systems from which you will be using data. This data has been exported ("dumped") in CSV format, which you can read in several ways. I would recommend using the String class's split() method to help you. You can view a CSV file in a spreadsheet or text editor to familiarise yourself with the data structure.

The first row of the CSV file contains column headers to help you understand what each field represents, so you will need to skip this line when reading. The remaining lines each contain the data values for a single plane, separated by commas. The fields in the file are as follows, and appear in the listed order:

Table 1: Aircraft CSV file fields

Field Name	**Description**
Tailcode	The aircraft's registration, as painted on the tail
Typecode	A four-letter string representing its make/model
Manufacturer	The assignment uses only Boeing and Airbus planes
Model	The plane's model number
Seats	How many passengers this plane can carry
CabinCrewRequired	How many cabin crew are needed to operate this aircraft
StartingPosition	The airport code of the plane's starting airport

An example row of data from the CSV file looks like:

```
G-DAJC,B767,Boeing,767,326,8,LGW
```

All of these fields map directly onto identically-named properties in the provided Aircraft class. You are tasked with completing the AircraftDAO (Data Accessor Object) class, which contains a method for loading data from the CSV file, and methods that can be used elsewhere in the system to acquire

one or more of the loaded Aircraft according to some specified criteria. Each of these methods is discussed, in turn, in Appendix A: Implementation Details at the end of this assignment specification.

The Employee Data (20 marks)

Data on the airline's employees is held on a system that can provide a snapshot of its data as a structured JSON file, which your code will read using the JSON reference parser you used in class. The file's format consists of a JSON object containing two keys: "pilots" and "cabincrew". Assigned to each of these keys is an array of JSON objects, each of which represents a single employee. An example of the JSON data, containing one pilot and one flight attendant, is depicted below.

```
{
  "pilots": [
    {
      "forename": "Brant",
      "surname": "Moots",
      "rank": "FIRST_OFFICER",
      "homebase": "EMA",
      "typeRatings": [
        "A320",
        "B757"
      ]
    }
  ]
  "cabincrew": [
    {
      "forename": "Shalon",
      "surname": "Hiser",
      "homebase": "BFS",
      "typeRatings": [
        "A320"
      ]
    },
  ]
}
```

Figure 3: Employee JSON Data Structure

Most of the fields in the figure above are shared between both cabin crew and pilots. The exception is the "rank" key, which is set only for pilots and will be set either as CAPTAIN or FIRST_OFFICER, which maps to the values specified in the Rank enum of the Pilot class in the assignment's base code. The "typeRatings" key lists each of the aircraft types a staff member is qualified to operate. Many staff members are qualified only for one aircraft type, although it is more common for cabin crew to be certified to operate a greater number of aircraft types as these qualifications are easier to obtain.

This information maps to various similarly-named properties in the Crew, Pilot and CabinCrew classes. It is the job of your DAO class to load this data from a JSON file formatted as above, and to provide various methods for finding crew members according to various criteria. Each of these methods is discussed, in turn, in Appendix A: Implementation Details at the end of this assignment specification.

The Route Data (20 marks)

Data on the airline's flight schedule: where the airline flies to and from, and when, is provided to you in an XML-based format. An example of the XML data, containing a single route, is depicted below.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<Routes>
    <Route>
        <FlightNumber>1848</FlightNumber>
        <DayOfWeek>Tue</DayOfWeek>
        <DepartureTime>13:40</DepartureTime>
        <DepartureAirport>Belfast</DepartureAirport>
        <DepartureAirportCode>BFS</DepartureAirportCode>
        <ArrivalTime>18:05</ArrivalTime>
        <ArrivalAirport>Tenerife-Sur</ArrivalAirport>
        <ArrivalAirportCode>TFS</ArrivalAirportCode>
        <Duration>PT4H25M</Duration>
    </Route>
</Routes>
```

Figure 4: Route XML Data Structure

Most of the fields in the XML data are self-explanatory. However, there are some notes and clarifications worth including here. Firstly, the DepartureTime and ArrivalTime elements use time local to the departure or arrival airport, respectively. As a result, adding the duration to the departure time of any flight that happens to cross time zones will *not* give the arrival time. Secondly, the Duration element uses an unusual-looking format to represent the times: The characters P and T, followed by the number of hours, followed by an H character, followed by the number of minutes, terminated by an M character. This is the format used by Java's java.time.Duration class, which contains a handy static parse() method for obtaining a Duration object from a string in this format.

This information maps to various similarly-named properties in the Route class. It is the job of your DAO class to load this data from an XML file formatted as depicted above, and to provide various methods for finding routes according to specified criteria. Each of these methods is discussed, in turn, in Appendix A: Implementation Details at the end of this assignment specification.

The Flight Booking Data (10 marks)

The flight booking data has been exported from the airline's booking system as an SQL dump of one of that system's database views, and imported into an SQLite database for use in the scheduling system. You will access this data using the xerial SQLite JDBC driver, as you used in class. The database's structure is simple, with just a single table containing three fields, as depicted below.

PassengerNumbers	
Date	TEXT
FlightNumber	INTEGER
Passengers	INTEGER

Figure 5: Database Table Structure

The table uses a compound primary key comprising the Date and FlightNumber fields. The table contains one row for each flight number on each of the days on which it will operate during July and August 2020, the period that we'll be working with for scheduling in the next section. Keep in mind that that there are almost 6800 flights in the data file loaded by the feedback generator, which could have performance implications if you architect your code on this section poorly.

The PassengerNumbersDAO does not rely too heavily on any of the provided classes. Instead, it (as previously) has a method for loading data from a SQLite database with the table as structured above, and some simple methods that allow the application to retrieve passenger forecasts for the specified flight and date. Each of these methods is discussed, in turn, in Appendix A: Implementation Details at the end of this assignment specification.

The Scheduler (15 marks)

This is the most challenging section of the assignment. It requires you to use data from all of the classes you wrote in the earlier sections to generate a schedule specifying which aircraft will be used on which route on which days, and which pilots and cabin crew will staff the flight. Your solution will be provided a start and end date, your DAOs for accessing aircraft, crew, route and forecast passenger number information. Schedulers are allowed a maximum of two minutes time to run for the purposes of this assignment, but in industry many similar tasks are performed over many hours overnight.

There are two different architectures that you can use when implementing your Scheduler. You could have the scheduler run for the full two minutes making multiple valid schedules and reporting its best so far to the caller, or you could implement it such that it makes only a single valid schedule when requested. To support both ways of working, your Scheduler will be run by a SchedulerRunner, included in the base classes. This will wait for a maximum of two minutes for your scheduler before calling its stop() method, but will return sooner if your scheduler returns its schedule before then. If you run for the full two minutes, it will be the last solution reported to the SchedulerRunner via its reportBestScheduleSoFar() method that is returned to the caller rather than that returned by the generateSchedule() method.

For a schedule to be valid, every flight in the time period must be allocated a plane and a full crew. A full crew is defined as a captain, first officer and at least the minimum number of cabin crew for the aircraft type (see the getRequiredCrew() method in the Aircraft class). In addition, no crew member or plane may be allocated to two flights at the same time (i.e. if any part of two flights overlap). You will not receive most of the marks on this section of the assignment unless the schedule you generate meets these two key requirements.

A valid schedule is not necessarily a good schedule. There's no way to know *a priori* if a schedule you generate is the best possible solution to a given set of constraints (e.g. staff and aircraft availability). Instead, we judge generated schedules by a set of objective criteria. We set out a list of *soft constraints*, which are undesirable characteristics that a schedule may exhibit, and assign each soft constraint a point score quantifying just how undesirable a characteristic is.

By counting how many times a schedule violates each of the soft constraints, and totalling up the points associated with each violation, we arrive at an objective measure of a schedule's quality. A lower score indicates a better schedule. The full list of soft constraints, and the quality score penalty points associated with each of them are presented in Appendix B, at the end of this assignment specification.

Hints & Tips

There is no maximum penalty score above which schedules will not be accepted in this section of the assignment. Marks will be awarded simply for a scheduler that generates any valid schedule. When writing your scheduler, there are some techniques that I would advise you use to help manage the complexity of the problem and your code.

1. Pair outbound and inbound flights together into a sensible shift, which you can then allocate a plane and the right number of crew to.

2. Remember to check the number of passengers expected for both flights in a pairing, otherwise you could end up with a large number of penalty points for leaving many passengers stranded abroad.

3. You do not have to schedule a plane and crew symmetrically: you could schedule Manchester-Ibiza-Gatwick to effectively move a plane into position for a future flight. Of course, the crew will now require some extra rest.

4. The order in which you make the allocations can significantly affect the complexity of your code, and the quality of your eventual schedule.

Competition (15 marks)

The final marks on this assessment will be allocated by way of a competition. All students whose work generates a valid schedule (i.e., with a plane and full crew scheduled to undertake each flight, and no plane/crew in two places at once) will be entered into the competition. In the competition, your scheduler will be allowed a maximum of *two minutes* to create the best schedule possible. Once two minutes have elapsed, your scheduler will be stopped, and the best schedule it has created will be assessed to generate a points score. Marks will be awarded as follows:

Table 2: Competition Prize Marks

Finishing Position	**Marks**
Bottom 50% of schedules	5 Marks
Top 50% of schedules	7 Marks
Top 25% of schedules	10 Marks
Top 10% of schedules	15 Marks

To give you a loose idea of what constitutes a reasonable point score for a generated schedule, I designed two algorithms for making schedules whilst writing the assignment. The first, which

generates truly terrible solutions, just picks aircraft and crew members randomly. These random schedules were given quality scores of around 2 billion. The second approach was an algorithm that attempted to make sensible decisions (allocating the right size plane, trying to allocate crew with the right qualifications etc.). Schedules generated by this algorithm were awarded a quality score of around 270 million. I'm sure some of you will be able to beat this!

First Steps

There is a small amount of starter code in the AircraftDAO and Main classes. The code in the AircraftDAO class serves as an illustration of how to read through and perform some simple processing on the aircraft CSV file. It prints out the tail code, type code and number of seats on each of the aircraft in the file. The code in the Main class runs the starter code in the AircraftDAO class, supplying it a Path to the aircraft CSV file in the project's data directory.

Once you have read through the rest of this specification, some sensible next steps would be:

1. Modify the AircraftDAO's loadAircraftData() method so that, for each row of the CSV file, it creates an Aircraft Object, using the setter methods to assign the values from the CSV row to its properties.

2. Modify the loadAircraftData() method again, so that all of the aircraft created whilst reading the CSV file are added to an ArrayList, which is then stored in an instance variable

3. Write the getAllAircraft() and the getNumberOfAircraft() methods, which are probably the simplest of the DAOs methods for retrieving aircraft.

Submission and Marking

Exporting Your Work

Your submission should consist of *both* the binary and source files of your implementation, zipped up into a single submission file. The easiest way to achieve this is using the File→Export menu in eclipse. On the export dialog screen, you can select General and then Archive File to export your work as a zip. On the next screen, you will need to select both the src and bin of your solution, *but not the provided base classes or libraries.* An example of the correct configuration is depicted below.

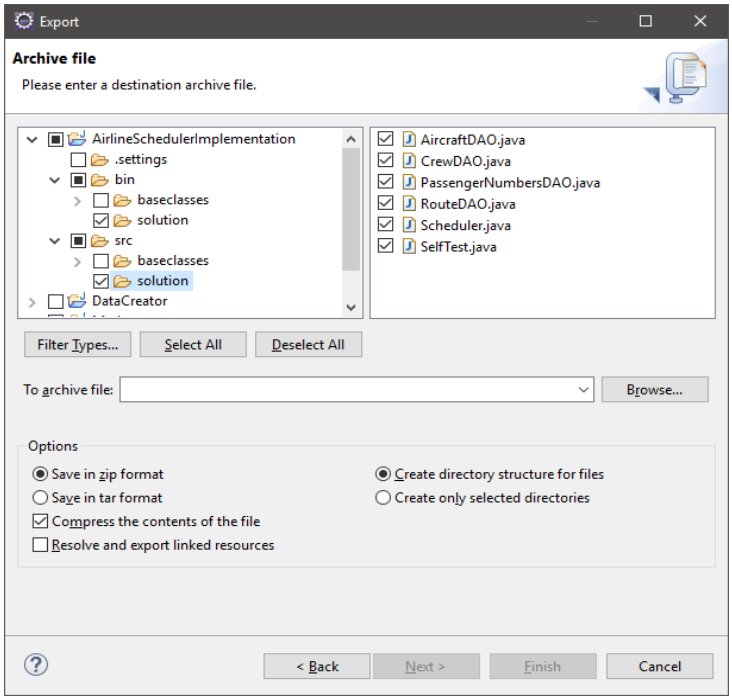

Figure 6: Eclipse Export Settings

Automatic Feedback

Included with the base classes and starter code for the assignment is an automatic feedback generator. This performs a number of unit tests on your work so far, and helps you to identify areas of functionality that are not working as specified, providing feedback as to which tests failed and why as a PDF, opened automatically when you run the generator.

This automatic feedback generator is a cut-down version of the program that will be used to mark the assignment when you submit it. Thus, it is important that you check your work against the feedback generator regularly. If the feedback generator cannot run your work, neither can the marker, and you'll be awarded a mark of zero for the assignment.

Formative Submission

At the end of the Autumn term, there will be a formative submission opportunity. Work submitted to the formative deadline will be run against a version of the automatic feedback generator that features a greater number of tests, giving you an opportunity to identify issues with your work that could cost you marks on the final submission. It also provides you with an opportunity to check that you're exporting your work correctly. Remember, work that cannot be run through the automatic marker will be awarded a mark of zero on this assignment, so it is *highly* recommended that you use the formative submission as a dry run.

Final (Summative) Submission

On the final, summative, submission, your work will be run through the most complete version of the automated marker. This will generate your mark and feedback for the assignment. Be sure that you

submit your work according to the instructions before the deadline. If your work cannot be marked, you will be awarded a mark of zero, which will seriously jeopardise your ability to pass the unit and will certainly hamper your mark for the Advanced Programming unit overall.

Marking Criteria

The automated marker will assess how correct your implementation of each of the completed sections is by running various unit tests upon your work. The marker will calculate your mark in each section of the work separately, according to the weights indicated in the section heading for each part of the work in this assignment brief. Each of the tests has an associated severity rating, which controls exactly how seriously you are penalised if your work fails such a test. The severity ratings, and the affect on your mark of failing tests of this severity, are listed below.

Minor It is expected that most students will be tripped up by a few of the minor tests along the way. As such 1-3 minor errors in a single section of the assignment will be penalised by a mark of 10% of the total mark for that section. If work fails more than 3 minor tests in a single section, it will instead be treated as a *significant* error.

Significant Significant errors indicate that a notable piece of the specified functionality isn't working as specified, but that this isn't enough to prevent the marking of the remainder of the section. Each significant error in a section of the assignment will be penalised by a mark of 25% of the total mark for that section.

Major Major errors indicate that some key functionality is not working. When the automated grader encounters a major error on a section, it abandons further testing on the section, awarding 25% of the total mark for that section as a final score, in recognition of some attempt having been made at the work. Furthermore, solutions containing any major errors are excluded from the scheduling competition, as the quality score calculated for schedules generated by these solutions may be unreliable.

Critical Critical errors are the most serious class of error, and indicate that a given section isn't working at all, or perhaps was not even attempted. If the automated grader encounters a critical error on a section, it abandons further testing and awards a mark of zero for the section. Furthermore, solutions containing any critical errors are excluded from the scheduling competition, as the quality score calculated for schedules generated by these solutions may be unreliable.

Plagiarism and Duplication of Material

I, the Faculty, and the University all take academic malpractice very seriously. The work you submit for this assignment must be your own, completed without any significant assistance from others. Be particularly careful when helping friends to avoid them producing work similar to your own. I will be running all submitted work through an automated plagiarism checker, and I am generally vigilant when marking. The penalties for academic malpractice can be severe. Please refer to the guidance at http://www.celt.mmu.ac.uk/plagiarism/ for further information.

Appendix A: Implementation Details

AircraftDAO

```
public void loadAircraftData(Path p) throws DataLoadingException
```

When supplied with a Path object pointing to a CSV file in the format discussed earlier, this method should cause the DAO to populate an internal data structure (e.g. an ArrayList) of Aircraft objects whose properties have been set to the same values as the corresponding columns in the CSV row. Multple calls to this methods should be additive: it should be possible to load more than one CSV file and have all the data from all files combined. The method should throw a DataLoadingException if there is a problem loading data from the file, for example if it is malformed.

```
public int getNumberOfAircraft()
```

This method should return the number of aircraft that have been loaded by the DAO.

```
public List<Aircraft> findAircraftByType(String typeCode)
```

When supplied with a type code (e.g. "B737"), this method should return a list of all of the loaded Aircraft objects of that type. If there are none, an empty list should be returned.

```
public List<Aircraft> findAircraftBySeats(int seats)
```

This method should return a list of all of the loaded Aircraft objects with **at least** the specified number of seats. If there are none, an empty list should be returned.

```
public List<Aircraft> findAircraftByStartingPosition(String startingPosition)
```

When supplied with a three letter airport code (e.g. "MAN"), this method should return a list of all of the loaded Aircraft objects that start at the specified airport. If there are none, an empty list should be returned.

```
public Aircraft findAircraftByTailCode(String tailCode)
```

When supplied with the short code, as painted on an aircraft's tail (e.g. "G-ABCD"), this method should return the loaded Aircraft object with the specified tail code. If no aircraft with this code has been loaded, this method should return null.

```
public List<Aircraft> getAllAircraft()
```

This method should return a list of all the currently loaded aircraft. If no aircraft have been loaded, it should return an empty list.

```
public void reset()
```

This method should unload all of the currently loaded aircraft, so that the DAO can be reset and new CSV files may be loaded.

CrewDAO

```
public void loadCrewData(Path p) throws DataLoadingException
```

When supplied with a Path object pointing to a JSON file structured in the manner presented earlier, this method should load all of the pilots and cabin crew contained within the file into appropriate internal data structures (e.g. an ArrayList), setting the properties of the appropriate objects correctly. As with the AircraftDAO, multiple calls to this methods should be additive. It should be possible to load the data from several files into a single DAO. If there is a problem loading the data, the method should throw a DataLoadingException.

```
public int getNumberOfPilots()
```

This method should return the number of pilots that have been loaded into the DAO.

```
public List<Pilot> getAllPilots()
```

This method should return a list of all of the pilots that have been loaded into the DAO. If none have yet been loaded, then an empty list should be returned.

```
public List<Pilot> findPilotsByTypeRating(String typeCode)
```

This method should return a list of all of the loaded pilots who are qualified to fly the specified aircraft type (e.g. “B737”). If no such pilot has been loaded, then an empty list should be returned.

```
public List<Pilot> findPilotsByHomeBase(String airportCode)
```

This method should return a list of all of the loaded pilots who are based at the airport with the specified airport code (e.g. “MAN”). If no such pilot has been loaded, then an empty list should be returned.

```
public List<Pilot> findPilotsByHomeBaseAndTypeRating(String typeCode, String airportCode)
```

This method should return a list of all of the loaded pilots who are *both* based at the airport with the specified airport code (e.g. “MAN”) *and* are qualified to fly the specified type of plane (e.g. “B737”). If there are no such pilots, an empty list should be returned.

```
public int getNumberOfCabinCrew()
```

This method should return the number of cabin crew that have been loaded into the DAO.

```
public List<CabinCrew> getAllCabinCrew()
```

This method should return a list of all of the cabin crew that have been loaded into the DAO. If none have yet been loaded, then an empty list should be returned.

```
public List<Crew> getAllCrew()
```

This method should return a single list of all the cabin crew *and* pilots that have been loaded into the DAO. The ordering of cabin crew vs pilots is unspecified, and code utilising this method must not be dependent on any particular ordering being delivered. If no crew (either pilots or cabin crew) have yet been loaded, this method should return an empty list.

```
public List<CabinCrew> findCabinCrewByTypeRating(String typeCode)
```

This method should return a list of all the loaded cabin crew who are qualified to steward flights on the specified aircraft type (e.g. "B737"). If no cabin crew qualified to fly this particular aircraft type have yet been loaded, then an empty list should be returned.

```
public List<CabinCrew> findCabinCrewByHomeBase(String airportCode)
```

This method should return a list of all of the loaded cabin crew who are based at the airport with the specified airport code (e.g. "MAN"). If no such cabin crew have yet been loaded, then an empty list should be returned.

```
public List<CabinCrew> findCabinCrewByHomeBaseAndTypeRating(String typeCode,
String airportCode)
```

This method should return a list of all of the loaded cabin crew who are *both* based at the specified airport (e.g. "MAN") *and* are qualified to steward flights on the specified aircraft type (e.g. "B737"). If no such cabin crew have yet been loaded, then an empty list should be returned.

```
public void reset()
```

This method should unload all of the currently loaded crew, so that the DAO can be reset and new JSON files may be loaded.

RouteDAO

```
public void loadRouteData(Path p) throws DataLoadingException
```

When supplied with a Path object pointing to an XML file formatted as specified earlier, this method should cause the DAO to load the routes contained within the file into an appropriate data structure (e.g. an ArrayList), setting the properties of the added route objects to appropriately reflect the values of the routes in the file. As with the previous DAOs, multiple calls to the loading method should be additive: it should be possible to load more than one file's worth of routes into a single DAO. If there is a problem loading the data, perhaps because a file is malformed, then a DataLoadingException should be thrown.

```
public int getNumberOfRoutes()
```

This method should return the number of routes that have been loaded into the DAO.

```
public List<Route> findRoutesByDayOfWeek(String dayOfWeek)
```

This method should return all of the routes that have been loaded which are specified to depart on the specified day of the week. The day of the week shall be supplied as a three-letter code (e.g. "Tue"). If no routes are specified to depart on the specified day of the week, then an empty list should be returned.

```
public List<Route> findRoutesbyDate(LocalDate date)
```

This method should return all of the routes that have been loaded which are specified to depart on the specified date (passed as a java.time.LocalDate object). I would recommend taking a look at LocalDate's getDayOfWeek() method to make the implementation of this method easier. If no routes are specified to depart on this date, then an empty list should be returned.

```
public List<Route> findRoutesDepartingAirport(String airportCode)
```

This method should return all of the loaded routes which depart the airport with the specified code (e.g. "MAN"). If no such routes have yet been loaded, then an empty list should be returned.

```
public List<Route> findRoutesByDepartureAirportAndDay(String airportCode, String
dayOfWeek)
```

This method should return all of the loaded routes which are *both* specified to depart the specified airport (e.g. "MAN") *and* depart on the supplied day (e.g. "Tue"). If no such routes have yet been loaded, then an empty list should be returned.

```
public List<Route> getAllRoutes()
```

This method should return a list of all of the currently loaded routes. If no routes have yet been loaded, then an empty list should be returned.

```
public void reset()
```

This method should unload all of the currently loaded routes, so that the DAO can be reset and new XML files may be loaded.

PassengerNumbersDAO

```
public void loadPassengerNumbersData(Path p) throws DataLoadingException
```

When supplied with a Path object pointing to an SQLite database structured as specified earlier, this method should load all of the projections into an appropriate data structure (e.g. a HashMap). It is recommended that you do not merely open a connection to the database and then run queries in getPassengerNumbersFor() and getNumberOfEntries(), as this will make it very difficult to enable multiple files of data to be loaded. This method, as with the previous DAOs should be capable of being called multiple times on multiple data sources, and should throw a DataLoadingException if there is a problem loading the data.

```
public int getPassengerNumbersFor(int flightNumber, LocalDate date)
```

This method should return the forecast number of passengers for the specified flight (e.g. 1234) on the specified date (passed as a java.time.LocalDate). If no forecast has been loaded for this flight and date pairing, then the method should return -1.

```
public int getNumberOfEntries()
```

This method should return the number of forecasts that have been loaded into the DAO.

```
public void reset()
```

This method should cause the DAO to unload all of the forecasts previously loaded, so that the DAO can be reset and new data may be loaded.

Scheduler

```
public void setSchedulerRunner(SchedulerRunner runner)
```

This method is called when setting up your scheduler. The scheduler is run by a SchedulerRunner, which reports the generated schedule back to the rest of the application. (and the marker!) If your scheduler is designed to run for the full two minutes, it will need to publish its best schedule so far via the ScheduleRunner's reportBestScheduleSoFar() method. After two minutes, the most recent published schedule will be returned by the SchedulerRunner. The SchedulerRunner will pass a reference to itself via this method for the scheduler to keep, and use to call reportBestScheduleSoFar() later. If you design your scheduler such that generateSchedule() reports a schedule back immediately upon completion, you do not need to implement any functionality in this method.

```
public Schedule generateSchedule(
      IAircraftDAO aircraftDAO,
      ICrewDAO crewDAO,
      IRouteDAO routeDAO,
      IPassengerNumbersDAO passengerNumbersDAO,
      LocalDate startDate,
      LocalDate endDate
)
```

This is the method responsible for generating a schedule. It is supplied with a copy of each of your DAO objects, each of which will have already loaded the relevant data. Also supplied are the dates (inclusive) at which the schedule should start and end. Your method will need to:

1. Create a Schedule object for the specified routes, start and end dates.

2. For each of the (so far blank) Schedule's FlightInfo objects, as returned by its getRemainingAllocations() method, allocate an aircraft, captain, first officer, and the right number of cabin crew found using the relevant methods in your DAOs. Call the Scheduler's completeAllocationFor() method for each FlightInfo method once done.

3. Once all flights have been allocated, return the completed schedule.

```
public void stop()
```

If your scheduler is designed to run for the full two minutes, then you should include a check in its main loop for a boolean variable to see if its time to stop. Have the scheduler set this variable when the stop() method is called. If you design your scheduler such that generateSchedule() reports a schedule back immediately upon completion, you do not need to implement any functionality in this method.

Appendix B: Schedule Quality (Soft Constraint) Calculation

The base classes include a QualityScoreCalculator class, which when supplied with an AircraftDAO, CrewDAO and PassengerNumbersDAO along with a Schedule, is capable of calculating a (potentially very large) points score reflecting the quality of a generated schedule. The bigger the point score, the worse a schedule is: we want to generate schedules with low scores! The automatic feedback generator will examine a schedule generated by your scheduler and total points based on the following criteria:

Passengers

50 points per passenger left behind because the plane allocated to their flight did not have enough seats to carry all of the booked passengers.

100 points if, in the previous scenario, the passenger was outside of the UK. It would be harder and more expensive to get them home.

All Staff

There are a significant number of complex rules governing the scheduling of airline crew. These rules are designed to prevent crew from being forced to fly tired, which is an obvious safety risk. The soft constraints listed here enforce a significantly simplified version of the rules, at the expense of the removal of some flexibility (e.g. crew flying out and back multiple times in one day).

10 points per hour worked by a staff member in excess of their monthly maximum (100 hours), calculated using calendar months, from the start of the scheduling horizon.

500 points if a crew member is allocated to a flight departing the UK from an airport other than their home base, and they were not first given at least 24 hours rest, calculated from when their previous flight landed (even if this was at the same airport they are now taking off from!)

500 points if a crew member is allocated to a flight landing in the UK at an airport other than their home base, and they were not given at least 24 hours rest afterwards, calculated using the scheduled departure time of their next flight (even if this is at the same airport at which they just landed!)

1,000 points if a crew member is allocated to a flight that departs from an airport outside the UK, unless they landed there in the past four hours.

5,000 points if, in the previous scenario, the crew member was not first given 48 hours of rest time (essentially, a day off plus another day to get to the right airport)

1,000 points if a crew member is allocated to a flight that lands at an airport outside the UK, but is not allocated to a return flight within the next four hours.

5,000 points if, in the scenario above, the crew member is not then given a period of 48 hours rest, allowing them to get home by alternative means, and then have a day off.

10,000 points if a crew member is not given a period of uninterrupted rest of at least 36 hours in any week, calculated on fixed 7-day windows from the start of the scheduling period.

20,000 points if a crew member is allocated to a flight that departs the UK within 12 hours of their last landing in the UK. Even if crew do not get a good night's rest, they should get something loosely equivalent.

50,000 points if a crew member is asked to crew a plane for which they are not qualified.

Pilots

These special rules govern the allocation of pilots to aircraft.

1,000 points if a flight is scheduled with a captain serving as first officer. Captains are more expensive than first officers, so this should be avoided wherever possible.

50,000 points if a flight is scheduled with a first officer serving as captain. This is obviously a far bigger problem than the other way round.

Aircraft

1 point per seat empty on a flight. Larger aircraft use more fuel, and airlines pay very close attention to their *load factor* (the percentage of the seats occupied on a flight). By applying a small penalty for empty seats, we incentivise schedules that are fuel-efficient.

100 points if an aircraft is scheduled to depart an airport within 30 minutes of landing there.

500 points if an aircraft is scheduled to depart an airport within 15 minutes of landing there.

20,000 points if an aircraft is scheduled to depart an airport other than that where it last landed. In reality, airlines can schedule a *positioning flight* to allow a plane to be moved from one airport to another. However, this would be incredibly complex for you to implement, so a fixed penalty is applied to represent the cost and logistical challenges of arranging such a flight offline.