THE UNIVERSITY OF QUEENSLAND
AUSTRALIA

School of ITEE
CSSE2002/7023 — Semester 1, 2021
Assignment 2
Due: 14 May 2021 16:00 AEST
Revision: 1.0.1

Abstract

The goal of this assignment is to implement and test a set of classes and interfaces[1], which build on the solution for the first assignment.

Language requirements: Java version 11, JUnit 4.

Please carefully read the Appendix A Document. It outlines critical mistakes which you must avoid in order to avoid losing marks. This is being heavily emphasised here because these are critical mistakes which must be avoided.

If at any point you are even slightly unsure, please check as soon as possible with course staff!

Preamble

All work on this assignment is to be your own individual work. As detailed in Lecture 1, code supplied by course staff (from this semester) is acceptable, but there are no other exceptions. You are expected to be familiar with "What not to do" from Lecture 1 and `https://www.itee.uq.edu.au/itee-student-misconduct-including-plagiarism`. If you have questions about what is acceptable, please ask course staff.

All times are given in *Australian Eastern Standard Time*. It is your responsibility to ensure that you adhere to this timezone for all assignment related matters. Please bear this in mind, especially if you are enrolled in the External offering and may be located in a different time zone.

Introduction

In this assignment you will finish building a simple simulation of an air traffic control (ATC) system.

In the first assignment you implemented the core model for the ATC. In the second assignment you will implement some of the more advanced logic to provide a very simple simulation for the ATC.

In this assignment, aircraft queues will be introduced. There are two types of aircraft queue; a landing queue and a takeoff queue. Aircraft can be added or removed from queues. It is also possible to check if an aircraft is in a queue, and to retrieve a list of all aircraft in the queue in order.

A control tower was introduced in the first assignment, but it had limited functionality. This will now be expanded. A control tower can undertake actions such as trying to land or takeoff aircraft, and placing aircraft in queues. The functionality for finding unoccupied gates will also be expanded from the first assignment.

[1]From now on, classes and interfaces will be shortened to simply "classes"

A control tower initialiser is also introduced in this assignment. The purpose of this is primarily to load information from 4 data files (which store information about aircraft and their tasks, terminals and gates, queues, and the number of ticks elapsed) into the ATC model when the program starts. This information will allow the simulation to be run, and for aircraft to move between various tasks.

Passenger and freight aircraft will now also have the ability to unload their passengers or cargo.

Multiple entities within the ATC now have the option to be encoded, and have hashcode and/or equals methods.

A simple GUI has been provided to you as part of the provided code. It is in the `towersim.display` package. *It will not work until you have implemented the other parts of the assignment that it uses.*

The GUI has been implemented using JavaFX and consists of three classes. `View` creates the main window for the ATC GUI. `AirportCanvas` displays the structure of the airport. `ViewModel` represents the ATC model that is to be displayed. The ATC application is initialised and started by the `Launcher` class in the `towersim` package. It loads the tick, aircraft, queue, and terminals and gates data and creates the GUI. Most of the GUI code has been provided to you.

In `ViewModel` you need to implement some of the logic that is executed by events in the simulation when buttons are pressed, and to save information in the ATC model to data files.

The functionality you need to implement in `ViewModel` is to:

- `ViewModel.getDroneAlertHandler()`.

  Defines what happens when the "Drone Alert" button is clicked.

  Declares a state of emergency on all terminals managed by the control tower.

- `ViewModel.getDroneClearHandler()`.

  Defines what happens when the "Clear Drone Alert" button is clicked.

  Clears the state of emergency on all terminals managed by the control tower.

- `ViewModel.getFindSuitableGateHandler()`.

  Defines what happens when the "Find Gate for Selected Aircraft" button is clicked.

  Updates information onscreen with the gate found for the currently selected aircraft.

- `ViewModel.saveAs()`.

  Saves the current state of the control tower simulation (which could then be loaded etc.).

Persistent Data & Loading Data from Files

Loading Data Files:
You need to implement loading the following from data files:

- The number of ticks elapsed.

  The JavaDoc for the `loadTick` method in the `ControlTowerInitialiser` class describes the format of a tick data file.

- Aircraft information.

  The JavaDoc for the `loadAircraft` method in the `ControlTowerInitialiser` class describes the format of an aircraft data file.

- Queue information.

  The JavaDoc for the `loadQueues` method in the `ControlTowerInitialiser` class describes the format of a queue data file.

- Terminals and gates information.

  The JavaDoc for the `loadTerminalsWithGates` method in the `ControlTowerInitialiser` class describes the format of a terminals and gates data file.

Example Data Files:
The following data files are provided. These show the **default** file implementations (when no information has been added).

- aircraft_default.txt
- queues_default.txt
- terminalsWithGates_default.txt
- tick_default.txt

The following data files are provided. These show **basic** implementations of the data files. Note that data files can contain more, or different information.

- aircraft_basic.txt
- queues_basic.txt
- terminalsWithGates_basic.txt
- tick_basic.txt

Supplied Material

- This task sheet.
- Code specification document (Javadoc).[2]
- A simple graphical user interface for the simulation, which is in the `display` package.
- A sample solution for the first assignment, available on Blackboard. You are to use this as the base for your implementation of the second assignment.
- A Subversion repositiory you can use for Version Control.

Javadoc

Code specifications are an important tool for developing code in collaboration with other people. Although assignments in this course are individual, they still aim to prepare you for writing code to a strict specification by providing a specification document (in Java, this is called Javadoc). You will need to implement the specification *precisely* as it is described in the specification document.

The Javadoc can be viewed in either of the two following ways:

1. Open `https://csse2002.uqcloud.net/assignment/2/` in your web browser. Note that this will only be the most recent version of the Javadoc.
2. Navigate to the relevant assignments folder under `Assessment` on Blackboard and you will be able to download the Javadoc `.zip` file containing html documentation. Unzip the bundle somewhere, and open `docs/index.html` with your web browser.

Tags in the Javadoc indicate what code has been implemented in assignment one and what code you need to implement in assignment two. Some code from assignment one will need to be modified. There are tags indicating places where you can expect to modify the assignment one code but these are not guaranteed to be all of the places where you may end up modifying code from assignment one.

[2]Detailed in the `Javadoc` section

Tasks

1. Implement the classes and methods described in the Javadoc as being required for assignment two.
2. Implement the indicated features of the user interface.
3. Write JUnit 4 tests for **all the methods in the following class**:
   - `LandingQueue` (in a class called `LandingQueueTest`)
4. Write JUnit 4 tests for **the following methods in the ControlTowerInitialiser class** (in a class called `ControlTowerInitialiserTest`). You do not need to write JUnit 4 tests for other methods in ControlTowerInitialiser (i.e. there will only be faulty implementations for these methods):
   - `loadAircraft`
   - `readAircraft`
   - `readTaskList`

Marking

The 100 marks available for the assignment will be divided as follows:

Symbol	**Marks**	**Marked**	**Description**
FT	45	Electronically	Functionality according to the specification
CF	5	Electronically	Conformance to the specification
SL	10	Electronically	Code Style: Structure and Layout
CR	20	By course staff	Code Style review: (Style and Design)
JU	20	Electronically	Whether JUnit tests identify and distinguish between correct and incorrect implementations

The overall assignment mark will be $A_1 = FT + CF + SL + CR + JU$ with the following adjustments:

1. If FT is 0, then the manual code style review will not be marked. CR will be automatically 0.
2. If SL is 0, then the manual code style review will not be marked. CR will be automatically 0.
3. If $SL + CR > FT$, then $SL + CR = FT$.

- For example: $FT = 22, CF = 8, SL = 7, CR = 18, J = 13$
  $\Rightarrow A_2 = 22 + 8 + (7 + 18) + 13$.

The reasoning here is to place emphasis on good quality functional code.

Well styled code that does not implement the required functionality is of no value in a project, consequently marks will not be given to well styled code that is not functional.

Functionality Marking

The number of functionality marks given will be

$$FT = \frac{\text{Unit Tests passed}}{\text{Total number of Unit Tests}} \cdot 45$$

Conformance

Conformance is marked starting with a mark of 5 (this is reduced compared to assignment 1). Every single occurrence of a conformance violation in your solution then results in a 1 mark deduction, down to a minimum of 0. Note that multiple conformance violations of the same type will each result in a 1 mark deduction.

Conformance violations include (but are not limited to):

- Placing files in incorrect directories.
- Incorrect package declarations at the top of files.
- Using modifiers on classes, methods and member variables that are different to those specified in the Javadoc. Modifiers include `private`, `protected`, `public`, `abstract`, `final`, and `static`. For example, declaring a method as public when it should be private.
- Adding extra public methods, constructors, member variables or classes that are not described in the Javadoc.
- Incorrect parameters and exceptions declared as thrown for constructors.
- Incorrect parameters, return type and exceptions declared as thrown for methods.
- Incorrect types of public fields.

Code Style

Code Structure and Layout

The Code Structure and Layout category is marked starting with a mark of 10.
Every single occurrence of a style violation in your solution, as detected by CheckStyle using the course-provided configuration[3], results in a 0.5 mark deduction, down to a minimum of 0. Note that multiple style violations of the same type will each result in a 0.5 mark deduction.

Note: There is a plugin available for IntelliJ which will highlight style violations in your code. Instructions for installing this plugin are available in the Java Programming Style Guide on Blackboard (Learning Resources → Guides). If you correctly use the plugin and follow the style requirements, it should be relatively straightforward to get high marks for this section.

Code Review

Your assignment will be style marked with respect to the course style guide, located under Learning Resources → Guides. The marks are broadly divided as follows:

Metric	**Marks Allocated**
Naming	6
Commenting	6
Readability	3
Code Design	5

Note that style marking does involve some aesthetic judgement (and the marker's aesthetic judgement is final).

Note that the plugin available for IntelliJ mentioned in the Code Structure and Layout section *cannot* tell you whether your code violates style guidelines for this section. To do so, it would need complex AI capabilities, which is not realistic. You will need to manually check your code against

[3]The latest version of the course CheckStyle configuration can be found at `http://csse2002.uqcloud.net/checkstyle.xml`. See the Style Guide for instructions.

the style guide.

The Code Review is marked starting with a mark of 20. Penalities are then applied where applicable, to a minimum of 0.

Metric	How it is marked

Naming	Misnamed variables (-6 marks max)
	e.g.
	→ Non-meaningful or one-letter names
	• `String temp; // bad naming`
	• `char a; // bad naming`
	• `int myVar, var, myVariable; // all bad naming`
	→ Variable names using Hungarian notation
	• `int roomInteger; // bad naming`
	• `List⟨Gate⟩ gateList; // bad naming`

Commenting	Javadoc comments lacking sufficient detail e.g. → Insufficient detail or non-meaningful Javadoc comments on (any) classes → Insufficient detail or non-meaningful Javadoc comments on (any) methods → Insufficient detail or non-meaningful Javadoc comments on (any) constructors → Insufficient detail or non-meaningful Javadoc comments on (any) class variables → etc. Lack of inline comments, or comments not meaningful e.g. → There needs to be sufficient comments which explain your code so that someone else can readily understand what is going. Someone should not need to guess or make assumptions. → Lack of inline comments, or comments not meaningful in methods → Lack of inline comments, or comments not meaningful in constructors → Lack of inline comments, or comments not meaningful for variables → etc.
Readability	Readability issues (-3 marks max) e.g. → Class content is laid out in a way which is not straightforward to follow → Methods are laid out in Classes or Interfaces in a way which is not straightforward to follow → Method content is laid out in a way which is not straightforward to follow → Variables are not placed in logical locations → etc.
Code Design	Code design issues (-5 marks max) e.g. → Using class member variables where local variables could be used → Duplicating sections of code instead of extracting into a private helper method → Using magic numbers without explanatory comments • `object.someMethod(50); // what does 50 mean? What is the unit/metric?`

JUnit Test Marking

See Appendix B for more details.

Assignment 1 had 16 faulty implementations. **Note that assignment 2 will have a higher**

number of faulty implementations..

The JUnit tests that you provide in `LandingQueueTest` and `ControlTowerInitialiserTest` will be used to test both correct *and* incorrect implementations of the `LandingQueue` and `ControlTowerInitialiser` classes. Marks will be awarded for test sets which distinguish between correct and incorrect implementations[4]. A test class which passes every implementation (or fails every implementation) will likely get a low mark. Marks will be rewarded for tests which pass or fail correctly.

There will be some limitations on your tests:

1. If your tests take more than 10 minutes to run (the lowest GradeScope can be set), or
2. If your tests consume more memory than is reasonable or are otherwise malicious,

then your tests will be stopped and a mark of zero given. These limits are very generous (e.g. your tests should not take anywhere near even 20 seconds to run).

Writing Tests for ControlTowerInitialiser

Due to the limitations of Gradescope, you may **not** include additional save files in your submission. This means that in order to write tests for ControlTowerInitialiser, you cannot create new save files and load them with FileReaders.

Instead, you must "embed" the contents of your custom save file into your ControlTowerInitialiserTest class by declaring the file contents as a String. Then, you can instantiate a StringReader and pass this reader into the methods in ControlTowerInitialiser. For example, to test that loading an invalid terminal and gates file throws a MalformedSaveException:

```
@Test
public void loadTerminalsWithGates_InvalidTest() throws IOException {
    String fileContents = String.join(System.lineSeparator(),
            "1",
            "AirplaneTerminal:notATerminalNumber:false:0" // invalid terminal number
    );
    try {
        ControlTowerInitialiser.loadTerminalsWithGates(
                new StringReader(fileContents), List.of());
        fail();
    } catch (MalformedSaveException expected) {}
}
```

Electronic Marking

The electronic aspects of the marking will be carried out in a Linux environment. The environment will not be running Windows, and neither IntelliJ nor Eclipse (or any other IDE) will be involved. OpenJDK 11 will be used to compile and execute your code and tests.

It is critical that your code compiles.

If your submission does not compile, **you will receive zero** for Functionality (FT).

Submission

How/Where to Submit

Submission is via Gradescope (submission is **not** via SVN repository like in previous semesters).

[4]And get them the right way around

GradeScope will be available for Assignment 2 **towards the end of Week 9**. You will not be able to submit your assignment before then (some things are still being checked by teaching staff so we cannot provide access yet).

You must ensure that you have submitted your code to Gradescope *before* the submission deadline. Code that is submitted after the deadline will **not** be marked (1 nanosecond late is still late).

What to Submit

Your submission should have the following internal structure:

`src/`	folders (packages) and `.java` files for classes described in the Javadoc
`test/`	folders (packages) and `.java` files for the JUnit test classes

A complete submission would look like:

```
src/towersim/Launcher.java

src/towersim/aircraft/Aircraft.java
src/towersim/aircraft/AircraftCharacteristics.java
src/towersim/aircraft/AircraftType.java
src/towersim/aircraft/FreightAircraft.java
src/towersim/aircraft/PassengerAircraft.java

src/towersim/control/AircraftQueue.java
src/towersim/control/ControlTower.java
src/towersim/control/ControlTowerInitialiser.java
src/towersim/control/LandingQueue.java
src/towersim/control/TakeoffQueue.java

src/towersim/display/AirportCanvas.java
src/towersim/display/View.java
src/towersim/display/ViewModel.java

src/towersim/ground/AirplaneTerminal.java
src/towersim/ground/Gate.java
src/towersim/ground/HelicopterTerminal.java
src/towersim/ground/Terminal.java

src/towersim/tasks/Task.java
src/towersim/tasks/TaskList.java
src/towersim/tasks/TaskType.java

src/towersim/util/EmergencyState.java
src/towersim/util/Encodable.java
src/towersim/util/MalformedSaveException.java
src/towersim/util/NoSpaceException.java
src/towersim/util/NoSuitableGateException.java
src/towersim/util/OccupancyLevel.java
src/towersim/util/Tickable.java

<<< must be test directory, not src! >>>
test/towersim/control/LandingQueueTest.java
test/towersim/control/ControlTowerInitialiserTest.java
```

Ensure that your classes and interfaces correctly declare the package they are within. For example, `Gate.java` should declare `package towersim.ground`.

Do not submit any other files (e.g. no `.class` files).
Note that `LandingQueueTest` and `ControlTowerInitialiserTest` will be compiled individually against a sample solution without the rest of your test files.

Provided set of unit tests

A small number of the unit tests (about 15-20%) used for assessing Functionality (FT) (not conformance, style, or JUnit tests) will be provided in Gradescope prior to the submision deadline, which you will be able to test your submission against.

The purpose of this is to provide you with an opportunity to receive feedback on whether the basic functionality of your classes is correct or not. Passing all the provided unit tests does **not** guarantee that you will pass all of the full set of unit tests used for functionality marking.

Late Submission

Assignments submitted after the submission deadline of 16:00 on May 14 2021 (by any amount of time), will receive a mark of zero unless an extension is granted as outlined in the Electronic Course Profile — see the Electronic Course Profile for details.

Do not wait until the last minute to submit the final version of your assignment. A submission that starts before 16:00 but finishes after 16:00 will not be marked. Exceptions cannot be made for individual students, as this would not be fair to all other students.

Assignment Extensions

All requests for extensions must be made via my.UQ as outlined in section 5.3 of the respective Electronic Course Profile. Please not directly email the course coordinator seeking an extension (you will be redirected to my.UQ).

Remark Requests

To submit a remark of this assignment please follow the information presented here:
`https://my.uq.edu.au/information-and-services/manage-my-program/exams-and-assessment/querying-result`.

Revisions

If it becomes necessary to correct or clarify the task sheet or Javadoc, a new version will be issued and an announcement will be made on the Blackboard course site.

Version 1.0.1 (Thu 29 April)

- Added section "Writing Tests for ControlTowerInitialiser" to provide instructions for writing unit tests when dealing with file I/O

Appendix A: Critical Mistakes which can cause loss in marks. Things you need to avoid!

This is being heavily emphasised here because these are critical mistakes which must be avoided.

The way assignments are marked has been heavily revised this semester to address many of these issues where possible, but there are still issues which can only be avoided by making sure the specification is followed correctly.

Code may run fine locally on your own computer in IntelliJ, but it is required that it also builds and runs correctly when it is marked with the electronic marking tool in Gradescope. Your solution needs to conform to the specification for this to occur.

Correctly reading specification requirements is a key objective for the course.

- Files must be in the exact correct directories specified by the Javadoc. If files are in incorrect directories (even slightly wrong), you may lose marks for functionality in these files because the implementation does not conform to the specification.
- Files must have the exact correct package declaration at the top of the file. If files have incorrect package declarations (even slightly wrong), you may lose marks for functionality in these files because the implementation does not conform to the specification.
- You must implement the public and protected members exactly as described in the supplied documentation (*no extra public/protected members or classes*). Creating public or protected data members in a class when it is not required will result in loss of marks, because the implementation does not conform to the specification.
  - Private members may be added at your own discretion.
- Never import the `org.junit.jupiter.api` package. This is from JUnit 5. This will automatically cause the marks for the JUnit section to be 0 because JUnit 5 functionality is not supported.
- Do NOT use any version of Java newer than 11 when writing your solution! If you accidentally use Java features which are only present in a version newer than 11, then your submission may fail to compile when marked. This will automatically cause the marks for associated files with this functionality to be 0.

Appendix B: How your JUnit unit tests are marked.

The JUnit tests you write for a class (e.g. LandingQueueTest.java) are evaluated by checking whether they can **distinguish between a correct implementation of the respective class** (e.g. LandingQueue.java) (made by the teaching staff), **and incorrect implementations of the respective class** (deliberately made by the teaching staff).

First, we run your unit tests (e.g. LandingQueueTest.java, ControlTowerInitialiserTest.java) against the correct implementation of the respective classes (e.g. LandingQueue.java, ControlTowerInitialiser.java).

We look at how many unit tests you have, and how many have passed. Let us imagine that you have 5 unit tests (**it should be more than this, 5 is just an example**) for TaskList.java and 4 unit tests (**it should be more than this, 4 is just an example**) for ControlTowerInitialiser.java, and they all pass (i.e. none result in Assert.fail() in JUnit4).

We will then run your unit tests in both classes (LandingQueueTest.java, ControlTowerInitialiserTest.java) against an incorrect solution implementation of the respective class (e.g. LandingQueue.java). For example, the `getAircraftInOrder()` method in the LandingQueue.java file is incorrect.

We then look at how many of your unit tests pass.

ControlTowerInitialiserTest.java should still pass 4 unit tests. However, we would expect that LandingQueueTest.java would pass **less than** 5 unit tests.

If this is the case, we know that your unit tests can identify that there is a problem with this specific implementation of LandingQueue.java.

This would get you <u>one</u> identified faulty implementation towards your JUnit mark.

The total marks you receive for JUnit are the correct number of identified faulty implementations, out of the total number of faulty implementations which the teaching staff create.

If your unit tests identified 60% of the faulty implementations, you would receive a mark of:
60% of 20 $\rightarrow$ 12/20.