School of ITEE
CSSE2002/7023 — Semester 2, 2021
Assignment 2
Due: 22 October 2021 16:00 AEST
Revision: 1.0.0

Abstract

The goal of this assignment is to implement and test a set of classes and interfaces[1], extending on the classes implemented in the first assignment.

Language requirements: Java version 11, JUnit 4.12.

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.

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!

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 continue developing a simple simulation of a cargo port system, building on the core model of the system you implemented in assignment one.

In assignment two, additional logic is added to the Port class to process the movements that are queued to be executed. Movements will now be actioned by the port when their action time is reached, moving cargo and ships in and out of the port.

The concept of time is introduced to the simulation. One tick in the simulation represents a minute in real life. The `Tickable` interface allows classes to specify behaviour that occurs on each simulation tick.

The `StatisticsEvaluator` abstract class provides a way to monitor various statistics relating to the port's operations. Four subclasses of `StatisticsEvaluator` are to be implemented, which gather and report data on the cargo and ships that have moved through the port. More subclasses could be implemented in future to provide more insights into the collected data.

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

Ships that are arriving at the port must wait in a `ShipQueue` before being allocated a quay to dock at. The `ShipQueue` prioritises inbound ships based on a set of rules that check each ship's current status. For example, ships carrying dangerous cargo must be docked before any other ship waiting in the queue.

To ensure that each `Cargo` and `Ship` instance must have a unique cargo ID and IMO number respectively, the concept of the cargo/ship registry is introduced. These registries are a global mapping of IDs/IMO numbers to `Cargo`/`Ship` instances, where duplicate keys are not allowed. Each new `Cargo`/`Ship` instance registers itself with its registry inside the respective constructor.

To facilitate saving and loading the state of the simulation to and from a file, the `Encodable` interface has been introduced. Any class implementing this interface must implement the `encode()` method, which encodes the current state of the class to a machine-readable string. Similarly, classes that can be encoded can also be decoded from a string to a class instance through their `fromString()` method.

Many classes now override the default `equals()` and `hashCode()` method implementations inherited from `Object`, allowing instances to be checked for equality according to their internal state.

A Graphical User Interface (GUI) using JavaFX has been provided in the `portsim.display` package. Note that the GUI will not work correctly until the other parts of the assignment it relies on have been implemented. The GUI consists of three classes: `View` is responsible for creating the visual elements displayed on screen, `PortCanvas` handles the graphics and drawing of the port in the main canvas, while `ViewModel` manages interaction between the View and the core classes of the model (e.g. `Port`). The `Launcher` class in the `portsim` package initialises the GUI and passes the save file to the `ViewModel`.

While most of the GUI code has been provided to you, several methods must be implemented to complete the GUI. These are the methods marked with "Implement this for assignment 2" in `ViewModel` – you do not (and should not) need to modify any code in `View` or `PortCanvas`.

Two examples have been provided of the save files used by the GUI to load the saved port simulation. These files are `saves/default.txt` and `saves/large.txt`, representing an empty port and a busy port with several ships, quays and queued movements respectively.

Supplied Material

- This task sheet.
- Code specification document (Javadoc).[2]
- Gradescope, a website where you will submit your assignment.[3]
- A starting template for your assignment code, available for download on Blackboard. The files in this template provide a minimal framework for you to work from, and build upon. These files have been provided so that you can avoid (some of) the critical mistakes described in Appendix A. Each of these files:
  - is in the correct directory (do not change this!)
  - has the correct package declaration at the top of the file (do not change this!)
  - has the correct public class or public interface declaration. Note that you may still need to make classes abstract, extend classes, implement interfaces etc., as detailed in the Javadoc specification.

[2] Detailed in the `Javadoc` section
[3] Detailed in the `Submission` section

As the first step in the assignment (after reading through the specifications) you should download the template code from Blackboard. Once you have created a new project from the files you have downloaded, you should start implementing the specification.

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 assignment 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 `doc/index.html` with your web browser.

Tasks

1. Fully implement each of the classes and interfaces described in the Javadoc.
2. Write JUnit 4 tests for all the methods in the following classes:
   - `ShipQueue` (in a class called `ShipQueueTest`)
   - `ShipThroughputEvaluator` (in a class called `ShipThroughputEvaluatorTest`)

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: Design review
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 $SL + CR$ will be capped at a maximum of FT.

- For example: $FT = 22, CF = 5, SL = 7, CR = 18, J = 13$
  $\Rightarrow A_1 = 22 + 5 + (7 + 18) + 13.$
  $\Rightarrow A_1 = 22 + 5 + (25) + 13.$ Limitation will now be applied.
  $\Rightarrow A_1 = 22 + 5 + (22) + 13.$

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.
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[4], 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.

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

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	4
Readability	3
Code Design	7

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. You will need to manually check your code against 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
	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> shipList; // bad naming ('ships' is better)`

Commenting	Javadoc comments lacking sufficient detail e.g. → Insufficient detail or non-meaningful Javadoc comments on classes, methods, constructors or 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, constructors, variables, etc.
Readability	Readability issues 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 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.

The JUnit tests that you provide in `ShipQueueTest` and `ShipThroughputEvaluatorTest` will be used to test both correct *and* incorrect implementations of the `ShipQueue` and `ShipThroughputEvaluator` classes. Marks will be awarded for test sets which distinguish between correct and incorrect implementations[5]. 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, 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 10 minutes to run).

[5] And get them the right way around

Writing Tests for `fromString` Methods

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 `ShipQueue.fromString()`, you cannot create new save files and load them with FileReaders.

Instead, you must "embed" the contents of your custom save file into your `ShipQueueTest` class by declaring the file contents as a string. Then, you can pass this string to `ShipQueue.fromString()`.

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.

Instructions for submitting to Gradescope will be made available on Blackboard (under Assessment → Assignment 2) within one to two weeks after the assignment specification has been released. You will not be able to submit your assignment before then.

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).

You may submit your assignment to Gradescope as many times as you wish before the due date, however only your last submission made before the due date will be marked.

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/portsim/Launcher.java

src/portsim/cargo/BulkCargo.java
src/portsim/cargo/BulkCargoType.java
src/portsim/cargo/Cargo.java
src/portsim/cargo/Container.java
src/portsim/cargo/ContainerType.java

src/portsim/display/PortCanvas.java
src/portsim/display/View.java
src/portsim/display/ViewModel.java

src/portsim/evaluators/CargoDecompositionEvaluator.java
src/portsim/evaluators/QuayOccupancyEvaluator.java
src/portsim/evaluators/ShipFlagEvaluator.java
src/portsim/evaluators/ShipThroughputEvaluator.java
src/portsim/evaluators/StatisticsEvaluator.java

src/portsim/movement/CargoMovement.java
src/portsim/movement/Movement.java
src/portsim/movement/MovementDirection.java
src/portsim/movement/ShipMovement.java

src/portsim/port/BulkQuay.java
src/portsim/port/ContainerQuay.java
src/portsim/port/Port.java
src/portsim/port/Quay.java
src/portsim/port/ShipQueue.java

src/portsim/ship/BulkCarrier.java
src/portsim/ship/ContainerShip.java
src/portsim/ship/NauticalFlag.java
src/portsim/ship/Ship.java

src/portsim/util/BadEncodingException.java
src/portsim/util/Encodable.java
src/portsim/util/NoSuchCargoException.java
src/portsim/util/NoSuchShipException.java
src/portsim/util/Tickable.java

test/portsim/evaluators/ShipThroughputEvaluatorTest.java
test/portsim/port/ShipQueueTest.java
```

Ensure that your classes and interfaces correctly declare the package they are within. For example, `Quay.java` should declare `package portsim.port`.

Do not submit any other files (e.g. no `.class` files).
Note that `ShipQueueTest` and `ShipThroughputEvaluatorTest` 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 10-20%) used for assessing Functionality (FT) (not conformance, style, or JUnit tests) will be provided in Gradescope prior to the submission deadline, which you will be able to test your submission against. In addition, a small number of the JUnit

faulty solutions used for assessing JUnit will be provided in Gradescope prior to the submission deadline.

The purpose of this is to provide you with an opportunity to receive feedback on whether the basic functionality of your classes and tests 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.

Instructions about the provided set of unit tests will be made available on Blackboard (under Assessment → Assignment 2) one or two weeks after the assignment specification has been released. Instructions will not be provided before then. This will still give you over two weeks to submit and check your work before the assignment is due.

Late Submission

Assignments submitted after the submission deadline of 16:00 on 22 October 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.

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. ShipQueueTest.java) are evaluated by checking whether they can **distinguish between a correct implementation of the respective class** (e.g. ShipQueue.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. ShipQueueTest.java) against the correct implementation of the respective classes (e.g. ShipQueue.java).

We look at how many unit tests you have, and how many have passed. Let us imagine that you have 7 unit tests in ShipQueueTest.java and 4 unit tests in ShipThroughputEvaluatorTest.java, and they all pass (i.e. none result in `Assert.fail()` in JUnit4).

We will then run your unit tests in both classes (ShipQueueTest.java, ShipThroughputEvaluatorTest.java) against an incorrect implementation of the respective class (e.g. ShipQueue.java). For example, the `peek()` method in the `ShipQueue.java` file is incorrect.

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

ShipThroughputEvaluatorTest.java should still pass 4 unit tests. However, we would expect that ShipQueueTest.java would pass **fewer than** 7 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 ShipQueue.java.

This would get you one 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.

For example, if your unit tests identified 60% of the faulty implementations, you would receive a mark of 60% of 15 $\rightarrow$ 9/15.