Object Oriented Programming (CSSE7023)

Assignment 2 — Semester 2, 2025

School of Electrical Engineering and Computer Science

The University of Queensland

Due October 24th 17:00 AEST

I hear and I forget, I see and I remember, I do and I understand.
— Confucian Aphorism

Revision 1.0

Overview Everybody loved your teletype game, but they realised they didn't specify it correctly! They actually wanted a GUI without all the scrolling and with buttons to click instead of typing in numbers. They also want to be able to save a game, so that the saved game can be re-loaded at a later time.

This assignment builds on Individual Assignment 1 and will extend your practical experience developing a Java program, including file I/O and a GUI. Additionally, you must develop and submit **JUnit 4** tests for some of the classes in the implementation. You are encouraged to write tests for *all* your classes and for the GUI part of your implementation as well, but you do not have to submit these. You will be assessed on your ability to

- implement a program that complies with the specification,
- develop JUnit tests that can detect bugs in class implementations,
- and develop code that conforms to the style conventions of the course.

Task - Adding a Graphical User Interface In this assignment, you will extend code **written by others**. You will utilise and extend a command-line version of the game written by other programmers, to extend the functionality to run the game in a Java Swing Graphical User Interface (GUI), as demonstrated in the Week 8 lecture. This Swing GUI version of the game is to be contained in a package `sttrswing`. The GUI must employ a *model-view-controller* (MVC) architecture.

- The "***model***" is the object-oriented representation of the thing being modeled – that is, the game logic and the internal representations of the information (the game objects, their state, and their methods) – this was the `sttr.game` package in *Individual Assignment 1* and now becomes the `sttrswing.model` package for *Individual Assignment 2*.
- The "***view***" is a visual representation of the *model*, retrieving data from the *model* to display to the user, and passing user input events to the *controller* and is packaged in the `sttrswing.view` package.
- The "***controller***" manages the *view(s)* and manipulates the *model* – it handles user input events, such as button presses or mouse movement and converts it to commands for the *model* or *view*. The *controller* is packaged in the `sttrswing.controller` package.

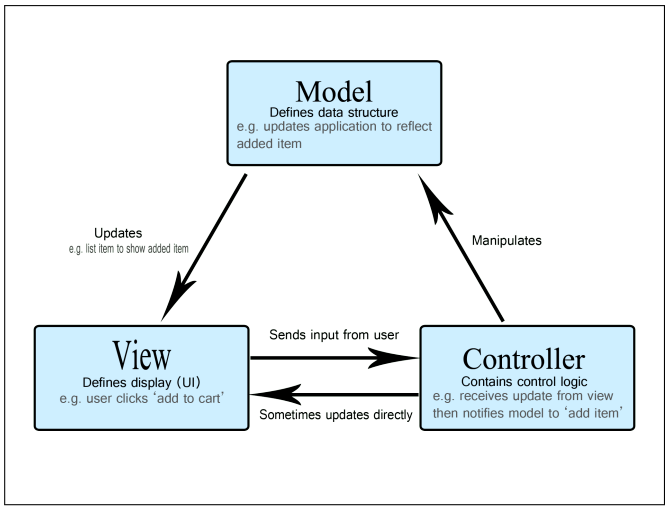

Figure 1: Mozilla Developer Network (MDN), model-view-controller-light-blue.png
`https://developer.mozilla.org/en-US/docs/Glossary/MVC/model-view-controller-light-blue.png`

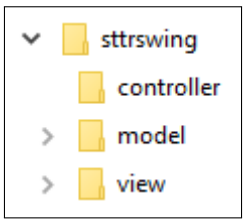

Figure 2: The top-level packages (*model* and *view* have sub-packages as well).

The **provided** *model* also includes the `sttrswing.model.validator` package (the same as Individual Assignment 1); and now also includes the interfaces in the `sttrswing.model.interfaces` package (previously removed from Individual Assignment 1 to simplify it – reinstated here so you can see the implementation), as well as a simple enum in the `sttrswing.model.enums` package to demonstrate enums.

For the *view* component you are **provided** with a colour palette (`Pallette.java`), a class that displays an image as an icon (`ImageLoader.java`), and the final views when the game is won or lost (`WinGameView.java` and `LoseGameView.java`). You are required to develop:

- the generic `View` class, as described in the Javadoc;
- the `StartView` class that provides the initial screen as described in the Javadoc;
- the `StandardLayoutView` class that is used throughout the gameplay as per Javadoc;
- and you must also populate two view sub-packages:
  - the `sttrswing.view.panels` package must contain **all** of the various panels that are used in the GUI layout during gameplay, as described in the Javadoc; and
  - the `sttrswing.view.guicomponents` package, which already contains two GUI elements that are frequently-used in the various panels (`DirectionButton.java` and `MapSquare.java`), requires you to develop the `Slider` GUI element as described in the Javadoc and demonstrated in multiple panels in the Week 8 lecture.

For the *controller* you are **provided** with an `ImageLoader` class (only used by the **provided** `LoseGameView` class – you do not need `ImageLoader` for any of your code), and you are required to develop:

- the `GameController` class, coordinates the interactions between the `model` and `views`, as described in the Javadoc;
- the `GameSaver` class that is used to save the current game state to a text file (described below); and
- the `GameLoader` class that is used to reload the previously saved game state (see below).

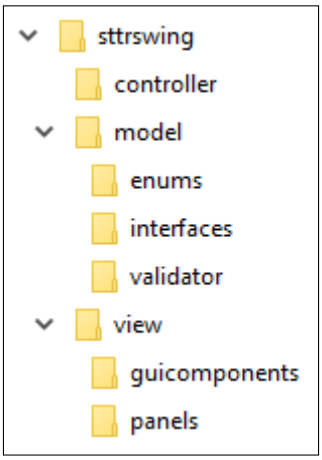

Figure 3: The complete heirarchy of packages.

Task - Adding a File menu with Save and Load menu items As part of the GUI (*view* and *controller*), you are required to develop a "`File`" menu, with "`Save`" and "`Load`" menu items, in order to save a game (just the Enterprise's current Quadrant and state, and all of the remaining numbers of Stars, Starbases, and Klingons for each Quadrant), so that the saved game can be re-loaded at a later time.

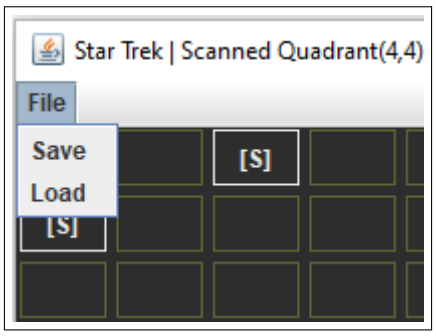

Figure 4: The "`File`" menu, with "`Save`" and "`Load`" menu items.

The saved game is always stored as a plain text file (the current state is stored as Strings only, not objects) in the `data` directory. The `data` directory must be directly under the root directory (i.e. at the same level as the `src` directory). You must not store (or load) the game from anywhere other than the `data` directory (using the current working directory will be marked incorrect).

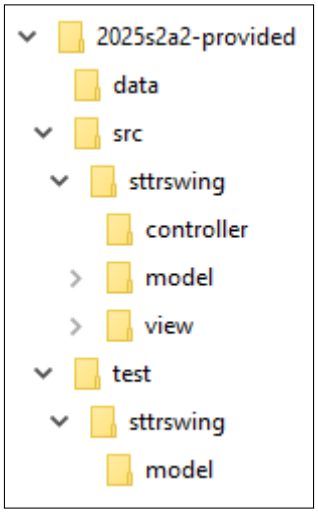

Figure 5: The top-level directories.

When re-loading the saved game, you must read and interpret (parse) input data (as Strings) to recreate the Quadrants with the numbers of Stars, Starbases, and Klingons they had when they were saved. Recreated Starbases and Klingons are brand new with their normal constructor parameters. While the recreated Quadrants have teir original saved numbers of Stars, Starbases, and Klingons, these will be in random positions as for any newly constructed quadrant.

Your implementation must write the saved game output data files in the correct format and will be tested to ensure you produce the specified format.

```
[e] x:5 y:6 e:2500 s:500 t:10 |
[q] x:0 y:0 s:111 |
[q] x:0 y:1 s:002 |
[q] x:0 y:2 s:110 |
[q] x:0 y:3 s:021 |
[q] x:0 y:4 s:022 |
[q] x:0 y:5 s:120 |
[q] x:0 y:6 s:012 |
[q] x:0 y:7 s:013 |
[q] x:1 y:0 s:112 |
[q] x:1 y:1 s:002 |
[q] x:1 y:2 s:211 |
[q] x:1 y:3 s:012 |
[q] x:1 y:4 s:101 |
[q] x:1 y:5 s:020 |
```

Figure 6: The `save.trek` file format.

Your submission will be tested against a number of input text files, not only the one your code writes – so your file output and input formats must match the specification, or those tests will fail.

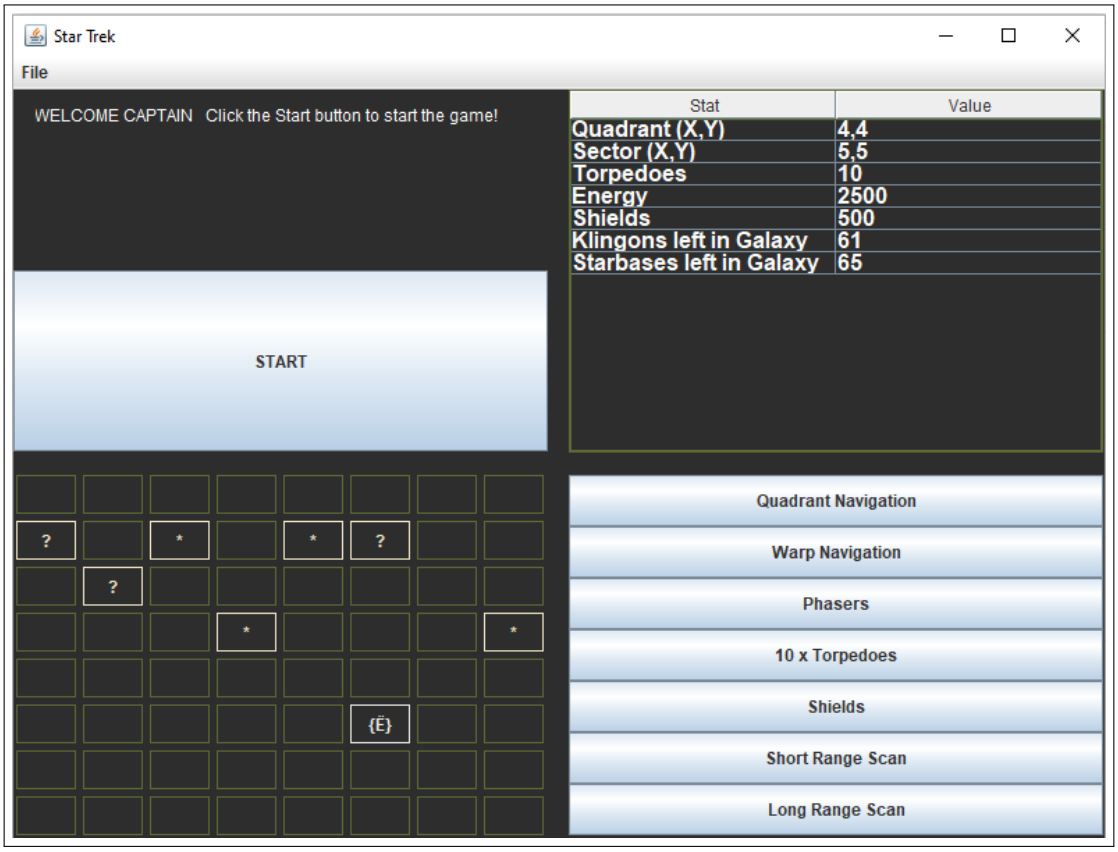

Figure 7: An image of the StartView view.

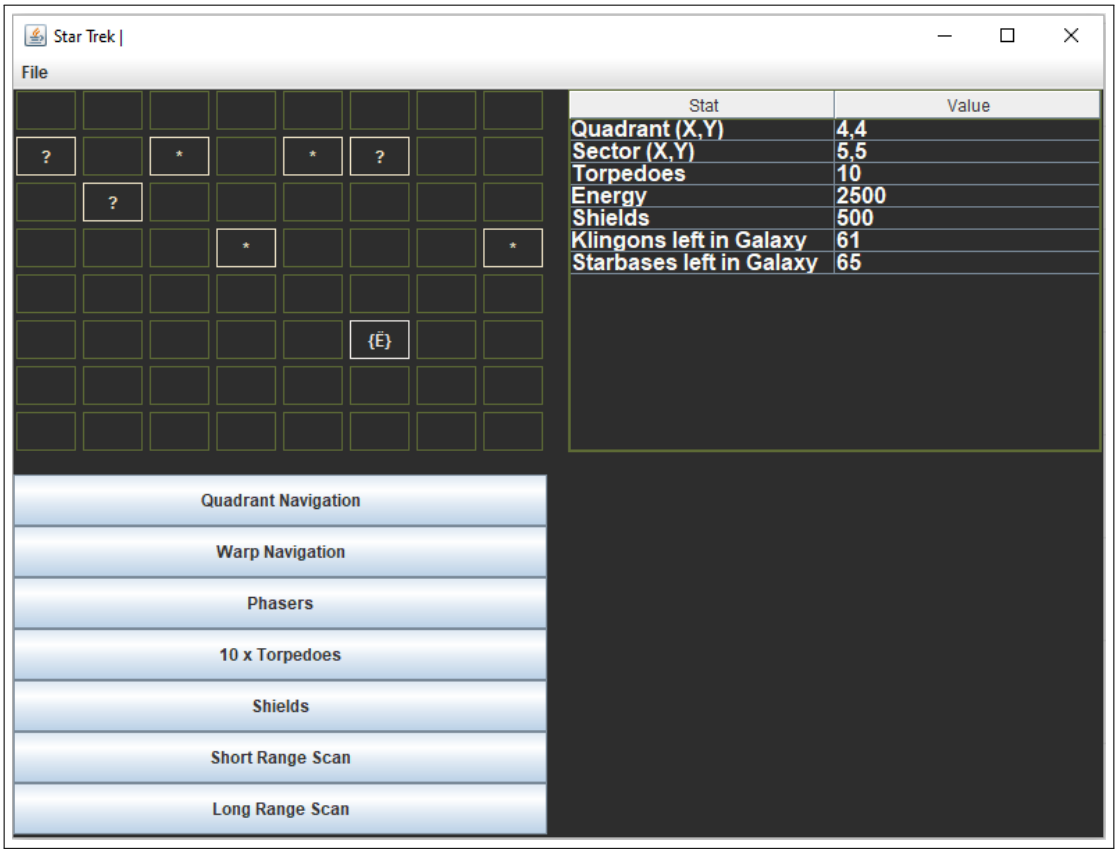

Figure 8: An image of the initial StandardLayoutView view.

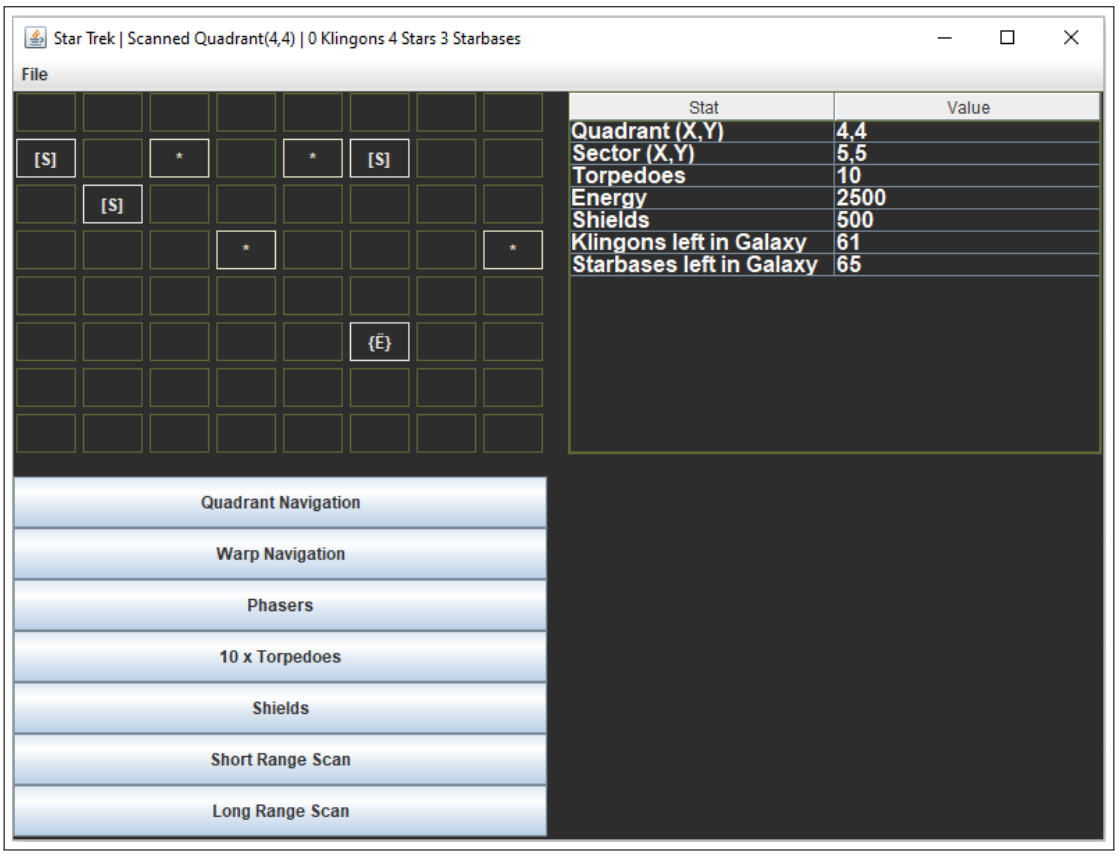

Figure 9: Short Range Scan – the StandardLayoutView **title** changes to show what happened.

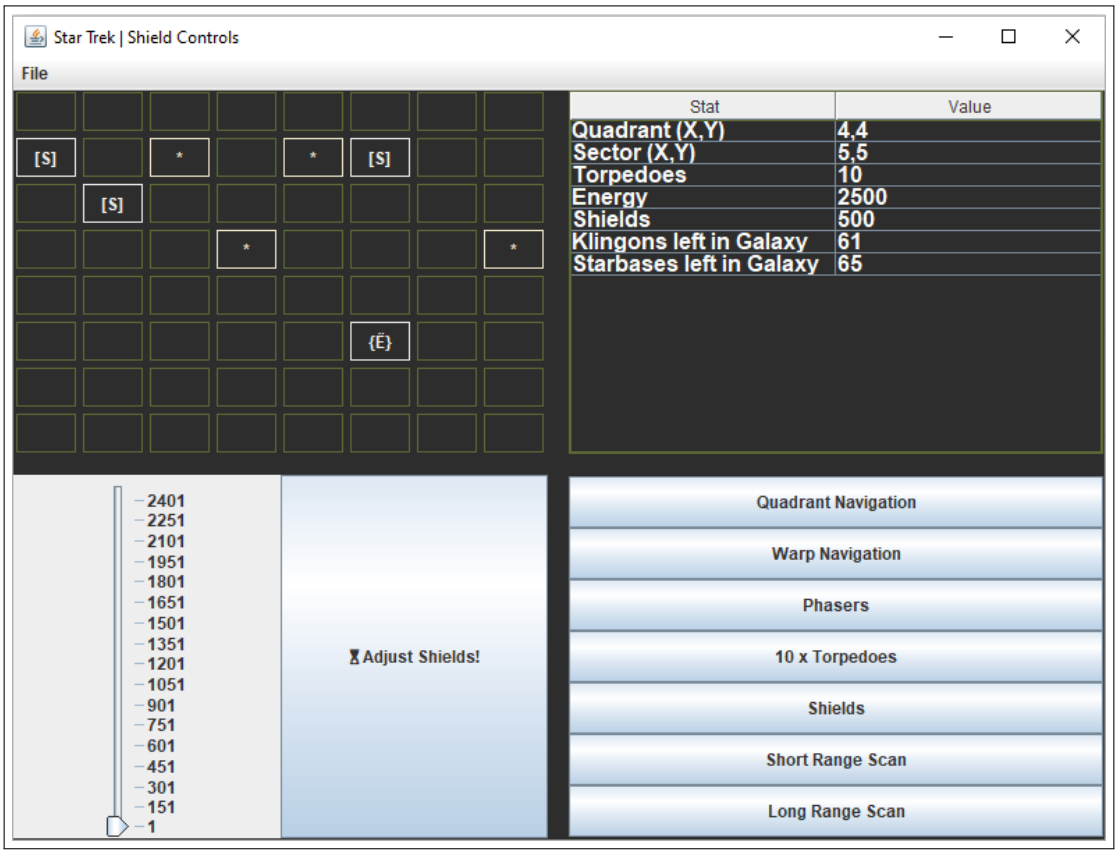

Figure 10: Shields – the Options panel moves to the right and the Shield panel appears on the left.

See the recording of the Week 8 lecture to view all the variations of the StandardLayoutView during gameplay.

Note, as stated in the Week 8 lecture, the somewhat ambiguous "Galaxy (X,Y)" and "Quadrant (X,Y)" labels that you see in the lecture recording, have been updated to "Quadrant (X,Y)" and "Sector (X,Y)" labels, that is we refer to *"Quadrant (X,Y)" within the Galaxy* and *"Sector (X,Y)" within the Quadrant*; and the hourglass symbols have also been removed from the two scan options (they were originally intended only as a reminder that scans, like everything else, take time and the enemy will attack you while you do it).

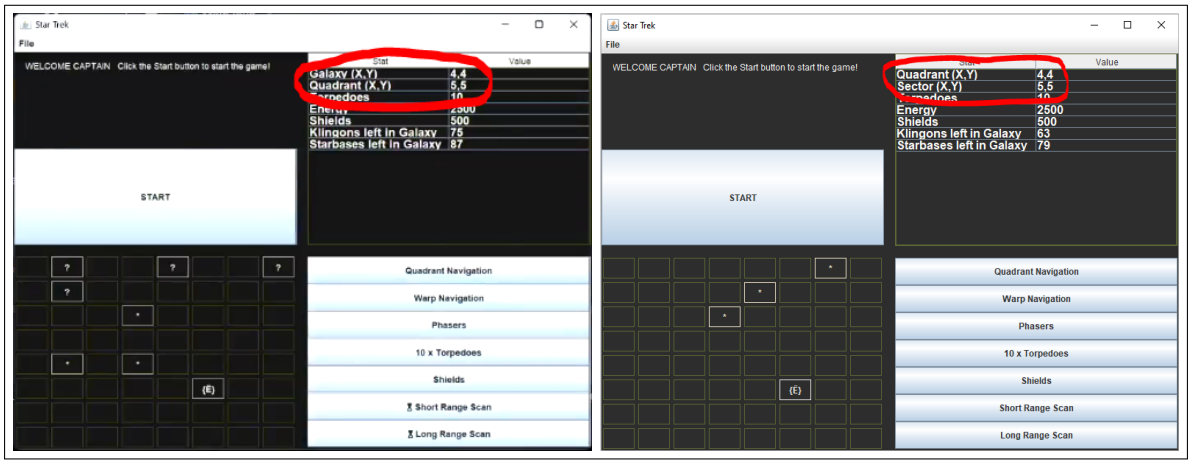

Figure 11: Now using our usual terminology: Quadrant within Galaxy and Sector within Quadrant.

Your new code will be tested against this specification and the supplied Javadoc. You are expected to test your code yourself. Any existing Gradescope tests are only to confirm that your submission compiles. The assessment tests will be conducted after all assignment submissions are complete (including all late submissions).

Task - JUnit Tests You will also write your own (assessed) tests for some of the classes in the implementation and include these in your submission. Only the tests for specific classes are to be submitted. Any test you write for other aspects of the project are not to be included in your submission.

The required tests that you have designed and submitted will be assessed for their effectiveness against correct implementations of the code, as well as a number of incorrect implementations specifically written by the staff to test your tests.

The required tests are for the `Entity`, `Klingon`, `Starbase`, `Enterprise`, and `Quadrant` classes. Your submitted test classes must be named `EntityTest`, `KlingonTest`, `StarbaseTest`, `EnterpriseTest`, and `QuadrantTest`. Each class may contain whatever tests you deem appropriate, but all the tests for each class must be in a single Java class file.

The Javadoc describes the classes and interfaces that your assignment must implement. A number of classes have not been included in the provided code. You must develop and submit new classes for these. **You should start with the provided code** and develop the new classes and methods described in the Individual Assignment 2 Javadoc as provided on Blackboard.

Use the `bundle` scripts included with the provided code to submit your complete assignment including your five test classes. **Note that the "`provided.zip`" includes the `bundle` scripts.**

Common Mistakes Please carefully read Appendix A. It outlines common and critical mistakes which you must avoid to prevent a loss of grades. If at any point you are even slightly unsure, please check as soon as possible with course staff.

Plagiarism All work on this assignment is to be your own individual work. By submitting the assignment you are claiming it is your own work. You *may* discuss the overall general design of the application with other students. Describing details of how you implement your design with another student is considered to be **collusion** and will be counted as plagiarism.

Do **not** show other students your code. Do **not** look at other students' code. Do **not** copy fragments of code from other students. Code supplied by course staff (from *this* semester) is acceptable, but must be clearly acknowledged as described in the next paragraph. You do not need to acknowledge the provided code that was supplied with this specification – it is part of the specification and is not included in the academic integrity checks. Code generated by third-party tools is also acceptable, but **must** also be clearly acknowledged, see *Generative Artificial Intelligence* below.

You may find ideas of how to solve problems in the assignment through external resources (e.g. StackOverflow, textbooks, ...). If you use these ideas in designing your solution you ***must*** cite them. To cite a resource, provide the full bibliographic reference for the resource in file called `refs.md`. The `refs.md` file **must** be in the root folder of your project. For example:

```
> cat refs.md
[1] E. W. Dijkstra, "Go To Statement Considered Harmful," Communications of the ACM,
    vol 11 no. 3, pp 147-148, Mar. 1968. Accessed: Sep. 6, 2025. [Online]. Available:
    https://www.cs.utexas.edu/users/EWD/transcriptions/EWD02xx/EWD215.html
[2] B. Liskov and J. V. Guttag, Program development in Java: abstraction,
    specification, and object-oriented design. Boston: Addison-Wesley, 2001.
[3] T. Hawtin, "String concatenation: concat() vs '+' operator," stackoverflow.com,
    Sep. 6, 2008. Accessed: Sep. 8, 2025. Available:
    https://stackoverflow.com/questions/47605/string-concatenation-concat-vs-operator
[4] ChatGPT-4, OpenAI, "How do I use invokeLater in Java Swing?",
    Accessed on: Sep. 19, 2025. [Online]. Available: https://chat.openai.com/
```

In the code where you use the idea, cite the reference in a comment. For example:

```java
/**
 * What method1 does.
 * [1] Used a method to avoid gotos in my logic.
 * [2] Algorithm based on section 6.4.
 */
public void method1() ...

/**
 * What method2 does.
 */
public void method2() {
    System.out.println("Some " + "content.") // [3] String concatenation using + operator.
}
```

You must be familiar with the university's policy on plagiarism. `https://uq.mu/rl553`
If you have questions about what is acceptable, please ask course staff.

Generative Artificial Intelligence Artificial Intelligence (AI) (e.g. ChatGPT or Copilot) and Machine Translation (MT) are emerging tools that may support students in completing this assessment task. Students may ***appropriately*** use AI and/or MT in completing this assessment task. Students ***must clearly reference*** any use of AI or MT ***in each instance***.

A failure to reference generative AI or MT use may constitute student misconduct under the Student Code of Conduct.

Please note that submitting an assignment that was written by AI, correctly referenced throughout, clearly identifying that all code was written by AI, is not plagiarism, but since the code was not written by the student, it ***has no academic merit*** and so would receive zero marks.

To pass this assessment, students may be required to demonstrate detailed comprehension of their submission independent of AI and MT tools, as part of the assignment marking process.

All submitted code will be subject to electronic plagiarism and collusion detection. The assignments are to be worked on individually and must be your own work except where the use of code written or provided by other entities is explicitly permitted by the assignment specification, ***and any such code is referenced in the manner required in the assignment specification.***

This is a learning exercise and you will harm your learning if you use AI tools inappropriately. Remember, you will be required to write code, by hand, in the final exam.

Specification

The specification document is provided in the form of Javadocs.

- Implement the classes and interfaces ***exactly*** as described in the Javadocs.
- Details not described in the Javadocs may be implemented as you see fit to produce the desired outcome.
- Read the Javadocs carefully and understand the specification ***before*** programming.
- Do not change the public specification in ***any*** way, ***including*** changing the names of, or adding additional, public classes, interfaces, methods, or fields.
- You are encouraged to add additional ***private*** members, classes, or interfaces as you see fit.
- JUnit 4 test cases that you submit must only test the public and protected methods as specified in the Javadocs. They must not rely on any other members, classes, or interfaces you may have added to the classes being tested. You may create private members and other classes in your test code.

You can download the Javadoc specification from **BlackBoard (Assessment → Individual Assignment 2)** or access it at the link below.

`https://csse7023.uqcloud.net/assessment/assign2/docs/`

Getting Started

To get started, download the provided code from **BlackBoard (Assessment → Individual Assignment 2)**. Extract the archive in a directory and open it with *IntelliJ*.

Grading

Five aspects of your solution will be considered in grading your submission:

1. *Automated functionality test*: the classes that you must implement will have a number of JUnit unit tests associated with them that we will use to test your implementation. The percentage of test cases that pass will be used as part of your grade calculation. Classes may be weighted differently depending on their complexity.
2. *JUnit test cases*: your JUnit test cases will be assessed by testing both correct *and* faulty implementations. The percentage of implementations that are appropriately identified as correct or faulty will be used as part of your grade calculation.
3. *Manual functionality test*: to ensure that the look and feel of your GUI implementation is the same as the original implementation, and to ensure that the `Load/Open` and `Save` menu options are properly invoked by the GUI, a small scenario with a number of steps in it will be manually executed by course staff. The faults identified during these steps will be used as part of your grade calculation.
4. *Automated style check*: Your grade for automated style checking is based on the number of style violations identified by the *Checkstyle* tool[1]. It will be run in the same environment as the JUnit automated functionality tests. Multiple style violations of the same type will ***each*** count as ***additional*** violations.

   Note: There is a plug-in available for *IntelliJ* that will highlight style violations in your code. Instructions for installing this plug-in are available in the Java Programming Style Guide on BlackBoard (Learning Resources → Guides). If you correctly use the plug-in and follow the style requirements, it should be relatively straightforward to get high grades for this section.
5. *Manal style check*: as for Assignment 1, the style and structure of your code will also be assessed by course staff. Your performance on these criteria will also be used as part of your grade calculation. It is therefore critically important that you read and understand the feedback for this part on Assignment 1, as soon as it is made available, so that you can address any issues in this assignment. See Appendix B for criteria that will be used to assess the readability of your code.

Appendix B shows how the above are combined to determine your grade for the assignment.

[1] The latest version of the course *Checkstyle* configuration can be found at `http://csse7023.uqcloud.net/checkstyle.xml`. See the *Style Guide* on BlackBoard for instructions on how to use it in *IntelliJ*.

Automated aspects of the assessment

Three aspects of assessment will performed automatically in a Linux environment: execution of JUnit test cases, running your JUnit test cases on correct and faulty implementations, and automated style check using *Checkstyle*. The environment will not be running Windows, and neither *IntelliJ* nor *Eclipse* (or any other IDE) will be involved. OpenJDK 21 with the JUnit 4 library will be used to compile and execute your code and tests. To prevent infinite loops, or malicious code, from slowing down Gradescope, any test that takes longer than 10 seconds to execute will be killed and identified as failing. All tests should execute in a small fraction of a second. Any test taking longer than a second to execute indicates faulty logic or malicious code. Similarly, any of your JUnit test cases that take longer than 20 seconds to execute on one of the correct/faulty implementations, or that consume more memory than is reasonable, will be stopped.

IDEs like *IntelliJ* provide code completion hints. When importing Java libraries they may suggest libraries that are not part of the standard library. These will ***not*** be available in the test environment and your code will ***not*** compile. When uploading your assignment to Gradescope, ***ensure*** that Gradescope says that your submission was compiled successfully.

Your code must compile.
If your submission does not compile, **you will receive no marks**.

Submission

Submission is via Gradescope. Submit your code to Gradescope *early and often*. Gradescope will give you some ***limited*** feedback on your code. Most importantly, it confirms that your code compiles and runs.
It will not do extensive testing before assessment, and it is ***not*** a substitute for testing your code yourself!

What to Submit Your submission ***must*** have the following internal structure:

`src/`	Folders (packages) and `.java` files for classes that you modified or created for this assignment.
`test/`	Folders (packages) and `.java` files for the JUnit tests that are required for this assignment.
`refs.md`	File containing the references for any citations in your code.

Included in the root directory of the provided code are the files `bundle.sh` and `bundle.cmd`. For MacOS and Unix users, run the `$bash ./bundle.sh` file to execute it. For Windows users, double-click or run the `.\bundle.cmd` file to execute it. This will create a `submission.zip` file for you to upload to Gradescope.

You can create the submission zip file yourself using a zip utility. If you do this, ***ensure*** that you do not ***miss*** any files or directories. Also ***ensure*** that you do not ***add*** any extra files. We recommend using the provided `bundle` scripts.

Ensure that your classes and interfaces ***correctly*** declare the package they are within. For example, `GameController.java` should declare `package sttrswing.controller;`.

Only submit the `src` and `test` folders and the `refs.md` file in the root directory of your project.
Do *not* submit ***any*** other files (e.g. no `.class` files or IDE files).

Provided tests A small number of unit tests will be provided in Gradescope to show your code compiled and can be tested. These are meant to provide you with an opportunity to receive feedback on whether the very basic functionality of your code works or not. Passing the provided unit tests does ***not*** guarantee that you will pass the tests used for functionality grading.

Assessment Policy

Late Submission You must submit your code ***before*** the deadline. Code that is submitted after the deadline will receive a late penalty as described in the course profile. The submission time is determined by the time recorded on the Gradescope server. A submission is not recorded as being received until uploading your files completes. Attempting to submit at the last minute may result in a late submission.

You may submit your assignment to Gradescope as many times as you wish before the due date. If a misconduct case is raised about your submission, a history of regular submissions to Gradescope, which demonstrate progress

on your solution, could support your argument that the work was your own.
However, that history may also show when a submission is plagiarised and subsequently changed to attempt to hide the plagiarism.
You are ***strongly*** encouraged to submit your assignment on time, or by the revised deadline if you have an extension. Experience has demonstrated that most students who submit their assignments late lose more grades due to the late penalties than they gain by making improvements to their work.

Extensions If an unavoidable disruption occurs (e.g. illness, family crisis, etc.) you should consider applying for an extension. Please refer to the following page for further information.

`https://uq.mu/rl551`

All requests for extensions must be made via my.UQ, ***before*** the submission deadline. Do not email the course coordinator or other course staff to request an extension.

Re-Grading If an *administrative error* has been made in the grading of your assignment, please contact the course coordinator (csse7023@uq.edu.au) to request this be fixed. For all other cases, please refer to the following page for further information.

`https://uq.mu/rl552`

Change Log

Revision: 1.0

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 course Blackboard site. All changes will be listed in this section of the task sheet.

A Critical Mistakes

THINGS YOU MUST AVOID

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

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 executed by the automated grading tool in Gradescope. Your solution needs to conform to the specification for this to occur.

- Files must be in the correct directories ***(exactly)*** as specified by the Javadoc. If files are in incorrect directories *(even **slightly** wrong)*, you may lose grades for functionality in these files because the implementation does not conform to the specification.
- Files must have the correct package declaration at the top of every file. If files have incorrect package declarations *(even **slightly** wrong, such as incorrect capitalisation)*, you may lose grades 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 specified will result in loss of grades, because the implementation does not conform to the specification.
    - You are *encouraged* to create private members as you see fit to implement the required functionality or improve the design of your solution.
- **Do not** change any of the provided code. Your submission must work with the provided code, as supplied. When you submit, any provided code is overwritten with a clean copy. If you accidentally change provided code in your working copy, you may find that your code only works on your machine and doesn't work when submitted.
- **Do not** import the `org.junit.jupiter.api` package. This is from JUnit 5 and may cause the JUnit tests to fail.
- Do not use ***any*** version of Java other than 21 when writing your solution. If you accidentally use Java features which are different in a version older than 21, then your submission may fail functionality tests. If you accidentally use Java features which are only present in a version newer than 21, then your submission may fail to compile.

B Grading

Overall Grade

Your overall grade will be calculated as a weighted percentage based on five components:

- Automated functionality testing will be performed by JUnit and count for 30%.
- Automated style checking will be performed by the *Checkstyle* tool and will be 10% (10 marks) ***LESS*** the number of style violations identified. Ten or more style violations will result in zero marks for this part.
- JUnit test marking will be performed by running your JUnit test cases against correct and faulty implementations and count for 20%.
- Manual functional testing will be performed by course staff and count for 20%.
- Manual Code Style and Structure testing will be performed by course staff, as described below, under `Code Style and Structure`, and count for 20%.

Code Style and Structure

The style and structure of your code will be assessed by course staff. Style and structure will be graded according to the provided rubric. The key consideration in grading your code style is whether the code is easy to understand. Code style will be assessed against the following criteria.

Readability

- Program Structure: Layout of code makes it easier to read and follow its logic. This includes using whitespace to highlight blocks of logic.
- Descriptive Identifier Names: Variable, constant, function, class and method names clearly describe what they represent in the program's logic. Do **not** use what is called the *Hungarian Notation* for identifiers. In short, this means do not include the identifier's type in its name (e.g. `item_list`), rather make the name meaningful. (e.g. Use `items`, where plural informs the reader it is a collection of items and it can easily be changed to be some other collection and not a list.)
- Named Constants: All non-trivial fixed values (literal constants) in the code are represented by descriptive named (symbolic) constants.

Documentation

- Comment Clarity: Comments provide meaningful descriptions of the code. They should not repeat what is already obvious by reading the code (e.g. `# Setting variable to 0.`). Comments should not be verbose or excessive, as this can make it difficult to follow the code.
- Complete Javadoc: Assignment 2 is after the Javadoc lecture and so is marked on **complete** Javadoc desciptions, including @param, @return, @link, and @throws tags. Every class, method, and member variable should have a Javadoc comment that explains its purpose. This includes describing parameters, return values, and potential exceptions so that others may understand how to use the method correctly.
- Description of Logic: All significant blocks of code should have a comment to explain how the logic works. For a small method, the logic should usually be clear from the code and Javadoc. For long or complex methods, each logical block should have an in-line comment describing its logic.

Design & Logic

- Single Instance of Logic: Blocks of code should not be duplicated in your program. Any code that needs to be used multiple times should be implemented as a method.
- Variable Scope: Variables should be declared locally in the method in which they are needed. Class variables are avoided, except where they simplify program logic.
- Control Structures: Logic is structured simply and clearly through good use of control structures (e.g. well-designed loops and conditional statements).
- Encapsulation: Classes are designed as self-contained entities with state and behaviour. Methods only directly access the state of the object on which they were invoked. Methods never update the state of another object.

Criteria	Standard		
Readability	**Advanced**	**Proficient**	**Developing**
Program Structure	Whitespace & comments highlight all blocks of logic, making it easy to follow.	Whitespace & comments highlight some blocks of logic, decreasing readability at times.	Whitespace & comments are not used well, decreasing readability in several places.
Identifier Names	All identifier names are informative and well chosen, increasing readability of the code.	Most identifier names are informative, aiding code readability to some extent.	Several identifier names are not informative, detracting from code readability.
Symbolic Constants	All, non-trivial, constant values are informative and well named, symbolic constants.	Most, non-trivial, constant values are informative, symbolic constants.	Only some, non-trivial, constant values are informative, symbolic constants.
Documentation			
Comment Clarity	Almost all comments enhance the comprehensibility of the code. Comments never repeat information already apparent in the code, nor are they verbose.	A few comments are unnecessary to code comprehension. Or, a few comments are overly verbose, reducing the ease with which code can be understood.	Many comments are unnecessary to code comprehension. Or, some comments are overly verbose, reducing the ease with which code can be understood.
Complete Javadoc	All provided Javadoc is replicated or improved. Complete & informative Javadoc is provided for all new methods & all member variables.	All provided Javadoc is at least replicated. Complete & informative Javadoc is provided for all new methods & most member variables.	Some Javadoc is inaccurate, unclear or absent.
Description of Logic	All important or complex blocks of logic are clearly explained or summarised. No stating of the obvious.	Most important or complex blocks of logic are clearly explained or summarised. Almost no stating of the obvious.	Some blocks of logic are poorly explained or summarised. Or, some descriptions are verbose or confusing.
Design & Logic			
Single Instance of Logic	Almost no duplicate code. Additional well designed methods modularise your code.	Some code has been duplicated. You have added some methods to modularise your code.	Large amounts of code are duplicated.
Variable Scope	All member variables are necessary parts of the class' abstraction. None should have been local to methods. Class variables have been avoided or simplify logic.	A few member variables should be local variables. Or, there are a few unnecessary local variables in methods. Class variables have been avoided.	Some member variables are unnecessary or should be local variables. Class variables have been used like modular global variables.
Control Structures	Logic is simple and clear through good use of control structures.	A small number of control structures are unnecessarily complex.	Some poorly designed control structures (e.g. excessive nesting or branching, overly complex logic, multiple unnecessary exit points, ...).
Encapsulation	All classes are independent entities with private state and public behaviour. Methods only directly access their own object's state, never modifying another's.	Most classes are are independent entities with private state and public behaviour. Methods rarely directly access or modify another object's state.	Some classes have non-private member variables. Some methods directly access or modify other objects' state.