Programming in the Large (CSSE2002)
Assignment 2 — Semester 1, 2025

School of EECS
The University of Queensland

Due May 14th 13:00 AEST

You can, if you want, rewrite forever.
— Neil Simon

Version 1.0.0

Overview

Greetings once again, Earthling! We, the governing body of Earth (not aliens), were so pleased with your implementation of the game ^{Not}*Space Invaders* that we have come to you once again to improve upon it. We have made a minor extension to the game; however, this code contains a number of problems and could use some help before we can play it properly. Please, Earthling, aid us in our quest for the perfect game!

This assignment provides experience working with an existing software project. Your task and its evaluation methods are described in this document.

Task

This assignment expands on the simple space shooter game from Assignment 1, asking you to integrate an Achievement system, fix certain identified limitations, and test your code. You will be assessed on your ability to:

- Identify and fix issues in the provided code;
- Develop robust JUnit tests that can detect bugs in class implementations;
- Implement changes to the functionality as described by the Javadocs, integrating new functionality in existing code with minimal disruptions;
- Write clear Java documentation *(Javadocs)*;
- Demonstrate good design, refactoring, and debugging practices; and
- Apply SOLID and clean coding principles through thoughtful refactoring.

Common Mistakes As with Assignment 1, the functionality of your code and its tests will be tested automatically. You should refer to Appendix A for a list of common mistakes to avoid.

Plagiarism All work on this assignment is to be your own individual work. Code supplied by course staff (from this semester) is acceptable, but must be clearly acknowledged. Code generated by third-party tools is acceptable but must be clearly acknowledged. See Generative Artificial Intelligence below. You must be familiar with the school policy on plagiarism:

`https://uq.mu/rl553`

Generative Artificial Intelligence You are strongly discouraged from using generative artificial intelligence (AI) tools to develop your assignment. 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 and justifications by hand in the final exam. If you do use AI tools, you must clearly acknowledge this in your submission. See Appendix C for details on how to acknowledge the use of generative AI tools. Even if acknowledged, you will need to be able to explain any part of your submission, so ensure that you fully undertstand the code that you are submitting.

Interviews In order to maintain assessment integrity and in accordance with the Assessment section of the course profile, interviews will be conducted in Weeks 12 and 13's practicals and contacts to evaluate genuine authorship and understanding of your assignment.

Further details are provided in the Components section below.

Software Design You will be given a broad specification of components and how they must be integrated with the existing program. The rest of the implementation design is up to you. You should use the software design principles that are taught in class (such as coupling, cohesion, information hiding, SOLID, etc.) to help your design.

Provided Code

The provided code and Javadocs can both be found on Blackboard.

Components

This assignment has six components. Each file component should be well-styled such that it is readable and understandable and should be well-designed such that it is extensible and maintainable.

1. Code Design: Do this throughout to ensure a high quality codebase.
2. Limitations: You need to identify and fix differences between the two assignments' Javadocs, improving upon various limitations from Assignment 1.
3. JUnit: You are asked to write JUnit tests for the limitations and their fixes.
4. Implementation: You are tasked with implementing the appropriate components to integrate an achievement system (including saving and loading where necessary) into the game.
5. Justification: Write your justification design document.
6. Interviews: These will be conducted in Weeks 12 and 13.

Component #1: Code Design

You should write your assignment code so that it aligns with SOLID principles as much as possible. This is primarily a design activity, and you are free to make sound design decisions that improve the code's cohesion, reduce coupling, and promote better information hiding. Your code should distribute responsibilities appropriately across the classes, making the system more modular and scalable.

Note that some elements of the Limitations component may require some refactoring.

The design of your classes is part of the assessment. Please be mindful that:

1. Discussing the design of your classes in detail with your peers may constitute collusion. You may discuss general design principles (cohesion, coupling, etc.) but avoid discussing your specific approach to this assignment.
2. Course staff will provide minimal assistance with design questions to avoid influencing your approach. You are encouraged to ask general software design questions.

Component #2: Limitations

While working on Assignment 1, you may have noticed a number of limitations of the game, such as collisions not registering properly, issues in the spawning logic, etc. Your task is to fix the 8–12 issues as specified in the provided Assignment 1 solution codebase and Assignment 2 Javadocs (treating these as bugs, not new features in need of implementation) and document your fixes in the justification document, clearly describing any new logic or changes. To do this, you may need to compare the Javadocs of Assignment 1 and Assignment 2. The Assignment 1 Javadocs are also provided on Blackboard.

Your task is as follows:

- Identify each limitation;
- Fix the code so that it matches the specification in the Javadoc; and
- Document your logic for each issue in the justification document. Include the reasoning, location, and a short rationale.

Note: Do not create additional public methods. You may, however, create new private helper methods where necessary.

Component #3: JUnit Tests

Write **JUnit 4** tests for the fixes implemented in the Limitations component. Your test cases should cover any methods you have changed, and are expected to be comprehensive enough to cover the full expected behaviour specified in the Javadocs, not just the differences from Assignment 1.

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

Never import the org.junit.jupiter.api package. This is from JUnit 5. This will cause the JUnit tests to fail, resulting in no marks for the JUnit component.

See the Marking section and Appendix B for more details.

Component #4: Implementation

You must also implement and integrate into the existing codebase an Achievement system, as defined by the Javadocs.

In this component there is also an element of file saving and loading. The file format is not specified. You must design an appropriate file format that is capable of storing the state. The saving and loading features must be compatible, and you must *not* utilise Java Serialization.

Integration should be achieved by making changes to `GameController` and `GameModel` to record player events and update achievements appropriately.

Component #5: Justification Document

While completing the above components, you need to compile a document detailing all the limitations you identified, how you fixed them, and also how your code complies with each of the SOLID principles (Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, and Dependency Inversion). You are free to use your own format for the document; however, it should clearly explain how your code complies with each of the SOLID principles.

Ensure that your document highlights how your changes improve the structure and maintainability of the codebase. You may use diagrams/code snippets as needed. All your justifications **MUST** refer to your codebase (i.e package → class → method ...), and they should not be generic descriptions.

In the assignment, you should ensure your justification document is concise, clearly explaining the thinking behind your design decisions. We've intentionally not set a word count to give you the space to express your ideas freely. However, avoid unnecessary length, and focus on the clarity of your explanations.

Your document should include the following content. However, you are free to add any additional sections if required, and you are welcome to write it with a different structure if you so choose.

1. Limitations
2. Fixes
3. Code design decisions (preferably organized according to each SOLID principle). Make sure to refer to your codebase to provide evidence and support your justifications.
4. Appendix (You must include your uses of GenAI here)

Component #6: Interviews

For the interview, you should be able to identify examples of code demonstrating SOLID principles in your assignment submission, in either the code you have written or in the sections provided to you. You may also be asked about topics including—but not limited to—fixes to the identified issues and test case rationale.

The interviews will be what counts towards the Design aspect of your Code Quality mark.

Tasks

1. Download the assignment `.zip` archive from Blackboard.
   - Import the project into IntelliJ.
   - Ensure that the project compiles and runs.
2. Identify and fix the errors in the provided code.
   - Compare the Javadocs of Assignments 1 and 2 to fix the limitations from Assignment 1.
   - The codebase contains several issues which are in need of fixing.
   - You must fix all errors in the code to implement the specification.
3. Ensure the code adheres to SOLID principles
   - This should be done at the same time as the Limitations and Implementation components.
   - Your code should follow SOLID principles, where possible, to ensure a good quality, scalable codebase that is able to be read, expanded upon, and maintained with minimal difficulty.
4. Write JUnit 4 tests for classes.
   - Write tests for the issues you fixed in the Limitations component.
   - Ensure the testing files are named correctly.
   - You should write your tests according to the specification.
5. Complete the implementation of the provided code.
   - The new components to be developed are for an Achievement system that has not been implemented.
   - You must implement these according to the specification in the Javadocs.
   - Note that all code you write must still adhere to good design principles.
6. Write a justification document on your design decisions.
   - List down the issues you identified and how you resolved them.
   - Provide justifications on design decisions you made in writing your code.
   - Clearly explain how your codebase complies with each of the SOLID principles.
   - **MUST** be a PDF file for submission.

Marking

The assignment will be marked out of 100. The marks will be divided into 5 categories: functionality (F), JUnit tests (T), bug fixes (B), code quality (Q), and style (S).

	Weight	*Description*
F	20	The program is functional with respect to the specification.
T	30	JUnit tests can distinguish between correct and incorrect implementations.
B	10	The errors in the provided code have been fixed.
Q	35	The new components have good code quality and the provided code has been refactored to improve its quality.
S	5	Code style conforms to course style guides.

The overall assignment mark is defined as

$$A_2 = (20 \times F) + (30 \times T) + (10 \times B) + (35 \times Q) + (5 \times S)$$

Functionality Each class has a number of unit tests associated with it on Gradescope. Your mark for functionality is based on the percentage of unit tests you pass. Assume that you are provided with 10 unit tests for a class, if you pass 8 of these tests, then you earn 80% of the marks for that class. Classes may be weighted differently depending on their complexity. Your mark for the functionality, F, is then the weighted average of the marks for each class,

$$F = \frac{\sum_{i=1}^{n} w_i \cdot \frac{p_i}{t_i}}{\sum_{i=1}^{n} w_i}$$

where n is the number of classes, w_i is the weight of class i, p_i is the number of tests that pass on class i, and t_i is the total number of tests for class i.

JUnit Tests The JUnit tests that you provide will be used to test both correct *and* incorrect (faulty) implementations of the game. Marks will be awarded for distinguishing between correct and incorrect implementations.[1] A test class which passes every implementation (or fails every implementation) will likely get a low mark.

Your tests will be run against a number of faulty implementations of the classes you are testing. Your mark for each faulty solution is binary based on whether at least one of your unit tests fails on the faulty solution, compared against the correct solution. For example, if you write 14 unit tests for a class, and 12 of those tests pass on the correct solution and 11 pass on a faulty solution, then your mark for that faulty solution is 1 (a pass). In general, if b_i of your unit tests pass on a correct implementation of class i and t_i pass on a faulty implementation, then your mark for that faulty solution is

$$f_i = \begin{cases} 1 & \text{if } b_i > t_i \\ 0 & \text{otherwise} \end{cases}$$

$$T = \frac{\sum_{i=1}^{n} f_i}{n}$$

where n is the number of faulty solutions.

See Appendix B for more details.

Bug Fixes For ease of reading, “bugs” here refers to the limitations of Assignment 1 that have been specified to be fixed in Assignment 2.

Each bug has a number of unit tests associated with it on Gradescope. Your mark for bug fixes is the percentage of bug-related unit tests you pass after fixing the bugs. For example, if the project has 40 unit tests associated with bugs and after fixing the bugs 30 tests pass, then you earn 75% of the marks for bug fixing. Your mark for bug fixing, B, is

$$B = \frac{p}{t}$$

where p is the number of unit tests that pass when you submit, and t is the total number of tests.

Code Quality The code quality of new features and refactored existing features will be manually marked by course staff.

To do well in this category of the marking criteria, you should consider the software design topics covered in this course. For example, consider the cohesion and coupling of your classes. Ensure that all classes appropriately document their invariants and pre/post-conditions. Consider whether SOLID principles can be applied to your software. The submitted code will be checked against the justification report when marking.

[1] And getting them the right way around.

An implementation with high code quality is one that is readable, understandable, maintainable, and extensible. The rubric on the following page details the criteria your implementation will be marked against. Ensure that you read the criteria prior to starting your implementation and read it again close to submission to ensure you meet the criteria.

The Design section will be marked via the interviews conducted after the due date, in which you may be asked questions about the principles, where they can be seen in the codebase, and other design decisions made throughout the assignment.

Readability (40%)

This will be evaluated manually by course staff through a careful review of the submitted code. Students shall ensure that these principles are consistently applied throughout their entire project (including the provided code).

Criteria	**Standard**				
	Advanced (100%)	**Functional (75%)**	**Developing (50%)**	**Beginning (25%)**	**No Evidence (0%)**
Method Decomposition (15%)	All functionality is neatly decomposed into small, coherent methods, each with a single clear purpose.	Most functionality is decomposed effectively, though a few methods may be too large or unclear.	Some methods are decomposed, but there are still instances of large or multipurpose methods.	There is at least one method that assume too many responsibilities.	Almost no attempt at method decomposition—code is mostly monolithic.
Documentation (5%)	All classes and public members have clear Javadoc comments, including usage examples and assumptions.	Most public members are documented; the overall purpose is fairly clear.	Public members are documented superficially; clarity or usage instructions may be incomplete.	Some documentation present, but it is incomplete or unhelpful.	No meaningful documentation.
Program Structure (10%)	Code within methods is well-organized, uses appropriate control structures, and uses spacing or comments to clarify logic. Complex blocks are minimized or well-documented.	Code is mostly structured and uses vertical spacing; control structures are generally suitable. Some complexity may be insufficiently explained. .	Code layout is somewhat inconsistent, with occasional unclear control flows or minimal spacing.	The code's structure often makes intent unclear; little attention is paid to organizing logic.	Code is disorganized and difficult to follow, with no consistent layout or patterns.
Descriptive Naming (10%)	All classes, members, and local variables have clear names that clarify their purpose	Most classes, members, and local variables have clear names that clarify their purpose.	Some classes, members, or local variables are named poorly and harm the readability of the code.	Minimal attempts have been made to create names that are clarifying	Almost no attempt has been made to create descriptive names for identifiers .

Design and Application of Programming Concepts (60%)

This will be evaluated through an interview, where students are expected to explain their design choices and demonstrate their understanding of each concept in the context of their project (including the provided code).

- **Information Hiding (20%)**
  Demonstrate understanding by appropriately protecting internal state and creating meaningful abstractions. The internal representation does not leak outside the class, ensuring that future changes to implementation can be easily managed.
- **Dependency Inversion (15%)**
  Demonstrate understanding by effectively using abstractions to invert program logic. Correct use of dependency injection enables easy modification of domain logic without extensive changes to existing code.
- **Polymorphism (15%)**
  Demonstrate understanding by effectively applying polymorphism to enhance program flexibility. Subclasses do not duplicate functionality from parent classes and consistently adhere to the substitution principle.
- **Contract Programming (10%)**
  Illustrate understanding by clearly documenting class invariants, pre-conditions, and post-conditions on public methods. Documentation clearly guides correct software behavior according to class contracts.

Code Style The Code Style category is marked starting with a mark of 5. Every occurrence of a style violation in your solution, as detected by *Checkstyle* using the course-provided configuration[2], results in a 1 mark deduction, down to a minimum of 0. For example, if your code has 2 checkstyle violations, then your mark for code quality is 3. Note that multiple style violations of the same type will each result in a 1 mark deduction.

$$S = \frac{max(0, 5 - v)}{5}$$

where v is the number of style violations in your submission.

Note: There is a plug-in available for *IntelliJ* which 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 marks for this section.

Electronic Marking

Marking will be carried out automatically in a Linux environment. The environment will not be running Windows, and neither IntelliJ nor Eclipse (or any other IDE) will be involved. Temurin-JDK 21 with the JUnit 4library will be used to compile and execute your code and tests. 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 zero marks**.

Submission

Submission is via Gradescope for the code and Blackboard for the Justification Document. Submit your code to Gradescope *early and often*. Gradescope will give you some feedback on your code, but it is not a substitute for testing your code yourself.

You must submit your code and document *before* the deadline. Anything that is submitted after the deadline will **not** be marked (1 nanosecond late is still late). See Assessment Policy.

You may submit your assignment to Gradescope and Blackboard as many times as you wish before the due date. Your last submission made before the due date will be marked.

Code Submission

Your code submission must include your code **(no .class, __MACOSX or .DS_Store files!)** in a zipped folder, just like in Assignment 1. This should include both your `src` and `test` folders. You should also include an `ai/README.txt` as per Appendix C below—if you have not used GenAI, you may leave this file empty, but it must still be submitted.

Ensure that your classes and interfaces correctly declare the package they are within. For example, `FileHandler.java` should declare `package game.achievements;`.

Justification Document

There is a separate submission link on Blackboard for the justification document. Submit your document in `PDF` format using the link. This link is **only** for the justification document, not for code.

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

Provided tests A small number of the unit tests used for assessing Functionality (F) will be provided in Gradescope, which can be used to test your submission against. In addition, a small number of the JUnit faulty solutions used for assessing JUnit tests (T) are also provided in Gradescope.

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 the tests used for functionality marking.

Generative Artificial Intelligence While the use of generative AI for this assignment is discouraged, if you do wish to use it, ensure that it is declared properly.

Code Generation

For this, you must create a new folder, called `ai/`, within this folder, create a file called `README.txt`. This file must explain and document how you have used AI tools. For example, if you have used ChatGPT, you must state this and provide a log of questions asked and answered using the tool. The `ai/README.txt` file must provide details on where the log of questions and answers are within your `ai/` folder. If you plan to use continuous AI tools such as Copilot, you must ensure that the tool is logging its suggestions so that the log can be uploaded. For example, in IntelliJ, you should enable the log by following this guide: `https://docs.github.com/en/copilot/troubleshooting-github-copilot/viewing-logs-for-github-copilot-in-your-environment` and submit the resulting log file.

For Justification Document

You are strictly prohibited from using generative AI for generating any portion of your justification document. If you choose to use it for grammar or language checks, you MUST include all the prompts utilized in an appendix

Assessment Policy

Late Submission Any submission made after the grace period (of one hour) will not be marked. Your last submission before the deadline will be marked.

Do not wait until the last minute to submit the final version of your assignment. A submission that starts before the end of the grace period but finishes after will not be marked.

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:

`http://uq.mu/rl551`

All requests for extensions must be made via my.UQ. Do not email your course coordinator or the demonstrators to request an extension.

Remarking If an *administrative error* has been made in the marking of your assignment (e.g. marks were incorrectly added up), please contact the course coordinator (csse2002@uq.edu.au) to request this be fixed.

For all other cases, please refer to the following page for further information:

`http://uq.mu/rl552`

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 marked with the electronic marking 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 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 members exactly as described in the supplied documentation (*no extra public members or classes*). Creating public members (instance variables or methods) in a class when it is not specified will result in loss of marks because the implementation does not conform to the specification.
    - You are encouraged to create private and protected members as you see fit to implement the required functionality or improve the design of your solution.
- Do not use any version of Java newer than 21 when writing your solution. If you accidentally use Java features which are only present in a version newer than 21, then your submission may fail to compile.

B JUnit Test Marking

The JUnit tests you write for a class (e.g. `BulletTest.java`) are evaluated by checking whether they can distinguish between a:

> **correct** implementation of the respective class
> e.g. `Bullet.java`, made by the teaching staff, and
>
> **incorrect** implementations of the respective class
> *deliberately made (sabotaged) by the teaching staff.*

First, we run your unit tests (e.g. `BulletTest.java`) against the correct implementation of the respective classes (e.g. `Bullet.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 `BulletTest.java` and 4 unit tests in `EnemyTest.java`, and they all pass (i.e. none result in `Assert.fail()` in JUnit4).

We will then run your unit tests in both classes (`BulletTest.java`, `EnemyTest.java`) against an incorrect implementation of the respective class (e.g. `Bullet.java`). For example, the `foo()` method in the `Bullet.java` file is incorrect.

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

`EnemyTest.java` should still pass 4 unit tests.
However, we would expect that `BulletTest.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 (incorrect) implementation of `Bullet.java`.

This would get you one identified faulty implementation towards your JUnit mark.

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

For example, if the teaching staff create 10 faulty implementations, and your unit tests identify 6 of them, you would receive 6 out of the 10 possible marks for JUnit, or 15 marks when scaled to 25%.

There are some limitations on your tests:

1. If your tests take more than 20 seconds 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 20 seconds to run).

C Generative Artificial Intelligence

While the use of generative AI for this assignment is discouraged, if you do wish to use it, ensure that it is declared properly.

For this, you must create a new folder, called “`ai`”. Within this folder, create a file called “`README.txt`”. This file must explain and document how you have used AI tools. For example, if you have used ChatGPT, you must state this and provide a log of questions asked and answered using the tool. The “`README.txt`” file must provide details on where the log of questions and answers are within your “`ai`” folder.

If you plan to use continuous AI tools such as Copilot, you must ensure that the tool is logging its suggestions so that the log can be uploaded. For example, in IntelliJ, you should enable the log by following this guide: `https://docs.github.com/en/copilot/troubleshooting-github-copilot/viewing-logs-for-github-copilot-in-your-environment` and submit the resulting log file.