Programming in the Large (CSSE2002)
Assignment 2 — Semester 2, 2024
School of EECS
The University of Queensland

Due October 15th 13:00 AEST

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

Version 1.0.0

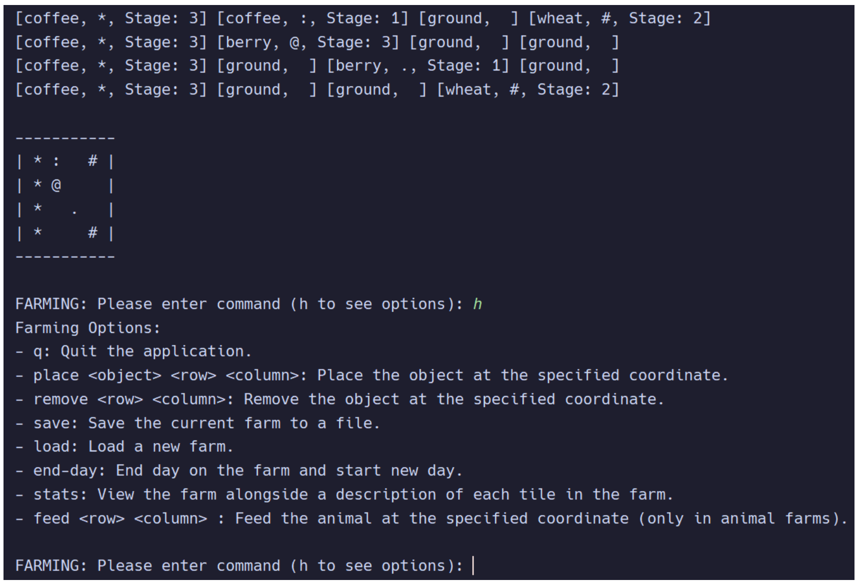
Overview This assignment provides experience working with an existing software project. You are provided with a small extension to the Farming game application. The extension has introduced errors and is poorly written. Fortunately, a suite of JUnit tests accompanies the extension. Additionally, you must write JUnit tests for a subset of previous classes.[1]

You will be assessed on your ability to:

- find and fix errors in provided code,
- meaningfully refactor provided code to improve its quality,
- extend the functionality of provided code, and
- write JUnit tests to a given specification.

Task Your program, Farm MVP, was a massive success! However, some users felt it was a bit boring, and needed more farming. You decide to make the product more exciting by introducing a new game to the Farm program.

The course staff have attempted to add a farming mini-game, in FarmGrid, but have made a real mess of it. Your task is to fix their mistakes and also refactor the code to improve its quality. Additionally you are expected to write JUnit tests for previous classes and extend the functionality of the new `FarmGrid` by adding saving and loading.



```
[coffee, *, Stage: 3] [coffee, :, Stage: 1] [ground,  ] [wheat, #, Stage: 2]
[coffee, *, Stage: 3] [berry, @, Stage: 3] [ground,  ] [ground,  ]
[coffee, *, Stage: 3] [ground,  ] [berry, ., Stage: 1] [ground,  ]
[coffee, *, Stage: 3] [ground,  ] [ground,  ] [wheat, #, Stage: 2]

-----------
| * :   # |
| * @     |
| *   .   |
| *     # |
-----------

FARMING: Please enter command (h to see options): h
Farming Options:
- q: Quit the application.
- place <object> <row> <column>: Place the object at the specified coordinate.
- remove <row> <column>: Remove the object at the specified coordinate.
- save: Save the current farm to a file.
- load: Load a new farm.
- end-day: End day on the farm and start new day.
- stats: View the farm alongside a description of each tile in the farm.
- feed <row> <column> : Feed the animal at the specified coordinate (only in animal farms).

FARMING: Please enter command (h to see options):
```

[1]Although you are encouraged to write tests for all your classes.

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. CCode 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, justifications by hand, in the final exam. If you do use AI tools, you must clearly acknowledge this in your submission. See Appendix C from A1 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.

Interviews In order to maintain assessment integrity and in accordance with section 5.4 of the course profile, you may be asked by the course coordinator via email to attend an interview to evaluate genuine authorship your assignment. Please refer to the course profile for further details.

Software Design In contrast to A1, you will not be given specification at the method level. Instead, you are given a broad specification of component(s) 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 (such as coupling, cohesion, information hiding, SOLID, etc.) that are taught in class to help your design.

Additional Features/Changes from A1

Grid An interface describing the methods and interactions on the farming grid. Contains the ability to:

- Interact with elements of the grid, such as feeding animals.
- Place things onto the Grid, such as crops to grow.
- Harvest things from the Grid
- Display the Grid's visual text representation of items.
- Get the stat information about each position on the Grid.
- Get the row and column number of the Grid.

This Grid interface is the key component of this new feature as it outlines how the FarmManager will interact with the new mini-games features. As such you **cannot** modify this interface in any way.

FarmGrid Represents the state of an animal or plant farm using a grid structure. It implements the Grid interface, providing methods for displaying and interacting with the grid.

The current responsibilities of the Farm Grid include:

- Storing farm information: FarmGrid stores the data representing the farm (e.g. plants, animals and empty spaces).
- Providing a visual representation: the farmDisplay() method from FarmGrid generates a string-based, visual representation of the farm.
- Manipulating farm information: Methods like place() from FarmGrid provide a way to interact with and change the farm information stored inside FarmGrid.

When it comes to refactoring `FarmGrid` you are free to make any modification you would like, including refactoring the entire class out if you so chose, as long as `FarmManager` still correctly stores some instance that implements `Grid`. Please note that if you rename `FarmGrid` or create new classes provided code may need to be updated to ensure everything is correctly instantiated. Make sure you are still passing all the tests for your class, not the original `FarmGrid`.

Farm Manager The behaviour of the FarmManager class is nearly identical to A1 apart from an additional farming mode. In the new farming mode, FarmManager will be responsible for coordinating interactions between the user and the farm model. It serves as the interface through which the user can view and modify the farm's state. By interpreting user input, the FarmManager allows the user to perform various actions that affect the farm, such as viewing the current layout and making changes to the farm's configuration.

The responsibilities of Farm Manager is nearly identical to A1 except for the following additions:

- Handling user input: FarmManager processes commands entered by the user, interpreting them and determining the appropriate actions to take based on the input.
- Displaying information: FarmManager provides the user with a way to see the current state of the farm by requesting and showing relevant data from the farm model.
- Updating the farm model: FarmManager ensures that the farm model is updated according to the user's actions, allowing for changes such as placing new items or modifying existing ones.
- Add products to inventory: FarmManager now allows items grown or collected from the farm to be added to the inventory to sell to customers as per the A1 functionality.

You are free to extend `FarmManager` as you wish as long as you leave the original logic flow and functionality intact. Make sure you are running the tests after you modify `FarmManager` to ensure they can still correctly access your `Grid`.

Components

This assignment has five components.

1. JUnit component: You are asked to write JUnit tests for a subset of previous classes.
2. Bug Fixing component: You need to debug `FarmGrid.java` to find and then fix bugs.
3. Refactoring component: Refactor the implementation of the `Farm Grid` to improve the design while maintaining its functionality.
4. Implementation component: You are required to implement the appropriate components that satisfies the specification on saving and loading.
5. Justification component: Write your justification design document.

Component #1: JUnit Tests

Write **JUnit 4** (Do not use JUnit 5 for the assignment) tests for all the public behaviour of the following classes:

- `BasicInventory`
  (in a class called `BasicInventoryTest`)
- `FancyInventory`
  (in a class called `FancyInventoryTest`)

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 A for more details.

Component #2: Debugging

The provided code comes with an implementation of `FarmGrid`. The new introduction of `FarmGrid` is not working as expected. Many tests of the given test suite will fail initially and this may be overwhelming. Find the smallest failing test — do not start with a complex test case as there are multiple bugs that may intersect. Create some theories about what may be wrong with the software; play testing the software may be helpful. You may find it helpful to construct some smaller test scenarios.

There are between 8 and 12 bugs, all of which are straightforward to fix with minimal changes to `FarmGrid.java`. You will need to have a clear understanding about what the bug is before attempting to fix it. You may find manually writing the test cases out helpful.
Note: It would be useful to detail the bugs in your justification document as you go!

You **must** complete the debugging before you begin the refactoring component. Not only will this make your life easier but you are required to copy the debugged version of `FarmGrid` into a new package before refactoring so we can easily identify your bug fixes without having to review the entire refactored code.

When the debugging is done copy `FarmGrid` from the package `farm.core.farmgrid` into a newly created package `farm.debugged.farmgrid`. The name of the class must remain as `FarmGrid` after copying. This should be a copy not a move, there should be two `FarmGrid` in your project as per below. See the submission section for more information.

```
farm/core/farmgrid/FarmGrid.java
farm/debugged/farmgrid/FarmGrid.java
```

Component #3: Refactoring

The current implementation of `FarmGrid` does not adhere to SOLID design principles. This makes the code difficult to maintain and extend.

In this component, your task is to refactor the `FarmGrid` class 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. The refactor should distribute responsibilities appropriately across new or existing classes, making the system more modular and scalable.

You are free to modify FarmGrid as much as you like including renaming it, as long as your new Grid/s implement the `Grid` interface and are correctly accessible through the `FarmManager` and are passing the tests. Re-read the *Additional Features/Changes from A1* section for more information.

Note: You should be modifying the `FarmGrid` class in the core package, not the one you copied over after debugging.

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.

You must preserve the existing functionality after bug fixing. To ensure that your implementation preserves the original functionality, JUnit tests have been developed. Your implementation should always pass these tests. To make this easy on yourself, ensure that you run the tests after each modification that may cause tests to fail.

Component #4: File Load & Saving

You must create two new features, one for saving a farm to a file and one to load a farm from a file. The file format is *not* specified. You must design a suitable file format that can store the state of a farm.

Saving and loading do not need to take into account the state of other areas of the game, such as the inventory, only the `Grid`. Loading is performed when the `FarmManager` is run, and saving can only be performed at the start of a new day, before any actions have taken place.

1. The saving and loading features **must** be compatible, i.e. saving a farm and then starting a new game and loading that farm should return the farm to that same state.
2. You must *not* utilize Java Serialization.
3. You must implement Saving and Loading inside the provided stub classes. You may add additional methods and add to existing ones, but do not change the signature of the provided method stubs.

To get you started the `FileSaver` and `FileLoader` classes have been created for you in the `farm.files` package with the minimum required stub methods.

Component #5: Justification Document

After completing the above 4 components, you need to compile a justifications document detailing your design decisions for refactoring the FarmGrid. It should also list all the bugs you identified and your fixes. You are free to use your own format for the document; however, it should clearly explain how your refactor complies with each of the SOLID principles (Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, and Dependency Inversion). Ensure that your document highlights how your changes improve the structure and maintainability of the FarmGrid component. You may use diagrams/code snippets as needed. All your justifications **MUST** refer to your codebase (i.e package → class → method ...) and it should not be generic descriptions.

In the assignment, you should ensure your justifications 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 sections. However, you are free to add any additional sections if required.

1. Justifications for Refactoring (preferably organized according to each SOLID principle). Make sure to refer to your codebase to provide evidence and support your justifications.
2. Bugs

3. Fixes
4. Appendix (You must include your uses of GenAI here)

Note that the provided tests are not a good demonstration of unit testing. As we want you to have a high degree of flexibility in how you implement your design, the tests are not granular and may be considered closer to integration tests.

Tasks

1. Download the assignment `.zip` archive from Blackboard.
   - Import the project into IntelliJ.
   - Ensure that the project compiles and runs (including running JUnit tests).
2. Identify and fix the errors/bugs in the provided code.
   - The `FarmGrid` component contains several bugs, causing the provided JUnit tests to fail.
   - You must fix all errors in the code to implement the specification and pass all the tests.
   - You must not change the provided tests in any way.
   - Copy your fixed `FarmGrid.java` class into the correct package once completed before refactoring.
3. Refactor the provided code.
   - The provided FarmGrid implementation is poorly implemented.
   - You must refactor the provided code to improve its quality.
   - You must not modify the behaviour of the original game while refactoring.
4. Complete the implementation of the provided code.
   - The components to be developed are for file saving and loading that have not been implemented.
   - You must implement these according to the specification above in the provided stub classes.
   - You are encouraged to create any additional classes that aid your implementation.
   - You are encouraged to write additional JUnit tests to test your new features.
5. Write a justification document on the design decisions.
   - Provide justifications on design decisions you made in refactoring the FarmGrid.
   - Clearly explain how your refactor complies with each of the SOLID principles
   - List down the bugs you identified and your fixes to them. You not need to explain the bug/fix, just what they were and how you resolved them.
   - **MUST** be a PDF file for submission.

Marking

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

	Weight	*Description*
B	10	The errors in the provided code have been fixed.
F	10	The provided code has been extended to include the specified new components and those components function as expected.
T	30	JUnit tests can distinguish between correct and incorrect implementations
Q	40	The new components have good code quality and the provided code has been refactored to improve its quality.
S	10	Code style conforms to course style guides.

The overall assignment mark is defined as

$$A_2 = (10 \times B) + (10 \times F) + (30 \times T) + (40 \times Q) + (10 \times S)$$

Bug Fixes The provided code includes JUnit tests that fail, indicating an incorrect implementation. You will be awarded marks for modifying the implementation such that the provided JUnit tests pass.

Your mark is based on the number of bugs you fix. The number of bugs you fix is determined by the number of unit tests you pass. For example, assume that the project has 40 unit tests and when given to you, 10 pass. After you have fixed the bugs, 25 tests pass. Then you have fixed 15 out of 30 bugs.[2] Your mark is then

$$\frac{25 - 10}{30} = \frac{15}{30} = 0.5 \ (50\%)$$

In general, let p_0 and f_0 be the number of unit tests that pass and fail in the provided code respectively. If p is the number of unit tests that pass when you submit, then your mark is

$$B = \max\left(\frac{p - p_0}{f_0}, 0\right)$$

Functionality Each class has a number of unit tests associated with it. 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 of the n classes,

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

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 in `BasicInventoryTest` and `FancyInventoryTest` will be used to test both correct *and* incorrect (faulty) implementations of the `BasicInventory` and `FancyInventory` classes. Marks will be awarded for distinguishing between correct and incorrect implementations.[3] A test class which passes every implementation (or fails every implementation)

[2] You do not get any marks for the unit tests that pass before you start and you cannot end up with a negative mark. If your modifications cause a test that originally passed to fail, that no longer counts as a pass.

[3] And getting them the right way around.

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 A for more details.

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

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.

Readability (40%)

Criteria	**Standard**				
	Advanced (100%)	**Functional (75%)**	**Developing (50%)**	**Little Evidence (25%)**	**No Evidence (0%)**
Method Decomposition (10%)	All functionality has been decomposed into small coherent methods that have singular clear purposes.	Most functionality has been decomposed into small coherent methods, however, some methods attempt to take responsibility for too much functionality.	The functionality of some methods is not clear as they are either extraneous or perform multiple tasks.	Methods are occasionally decomposed appropriately, however, on the whole most methods perform too many tasks.	Almost no attempt has been made to decompose methods.
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 .
Documentation (10%)	All classes and public members have clear Javadoc comments that explain how to utilize the classes and members. Documentation may include usage examples.	All classes and public members have Javadoc that attempts to explain how classes and members should be utilized. However, the documentation does not make the purpose and/or expected usage of the software obvious.	All classes and public members have Javadoc comments but they occasionally do not help to explain how to utilize the class in a meaningful way.	Not all classes and public members have Javadoc comments or documentation is not helpful.	Minimal evidence or no evidence of Javadoc documentation throughout the submission.
Program Structure (10%)	The structure of code within methods is clear. Vertical spacing is utilized to separate any logical blocks of code. The control structures are suitable for the task. Any complex blocks or lines of code have been minimized and are appropriately documented.	The structure of code within methods is appropriate but not clear. Vertical spacing is mostly utilized to separate any logical blocks of code. Control structures are somewhat convoluted but mostly appropriate. Any complex blocks or lines of code have been minimized or are appropriately documented.	The structure of code within methods occasionally makes it difficult to understand the intention of the code. An attempt has been made to structure the code but falls short of being well structured.	The structure of code within methods makes the intention of the code or functionality of the method difficult to understand.	The structure of code within methods is poorly structured and hard to read.

Design (60%)

Criteria	**Standard**				
	Advanced (100%)	**Functional (75%)**	**Developing (50%)**	**Little Evidence (25%)**	**No Evidence (0%)**
Information Hiding (15%)	All classes hide and protect their internal representation. References to internal state is protected by appropriate copying. An appropriate abstraction has been created around the internal representation that would make it feasible to later change implementations.	All classes attempt to hide and protect their internal representation, however, some leakage of internal state may occur. Consideration has been made towards creating an appropriate data abstraction.	Most classes have attempted to protect their internal representation. Some methods that should be private are public, damaging the data abstraction.	Minimal attempts have been made to protect the internal representation of classes. Methods that should be private are public or member variables have been made public.	Member variables/methods are public in a way that encourages tight coupling. No or minimal evidence of data abstraction.
Dependency Inversion (15%)	The logic of the program is inverted in a way that makes most classes highly configurable. The domain logic of the program is passed down allowing easy future modification.	It is possible to alter some of the program logic via dependency injection. However, non-trival changes would mostly still require modification.	An attempt has been made to parameterise classes by abstractions of domain logic. However, the abstractions are not at an appropriate level.	No attempt has been made to enable changing the program logic by dependency inversion. Some attempt has been made to parameterise classes but program logic is rigid.	The software is highly rigid, any alterations of the program logic would require modification of low-level program components.
Cohesion (10%)	All classes are highly cohesive with each serving a clear single-minded purpose.	Most classes are highly cohesive, serving a clear purpose.	There are no “God classes” but not all classes are highly cohesive.	There is at least one “God class” that assumes too many responsibilities.	No/minimal attempt has been made to decompose classes such that each is cohesive.
Polymorphism (10%)	Polymorphism is used to enhance the flexibility of the program. No subclasses duplicate the functionality of the parent. No classes violate the substitution principle.	Polymorphism is used and no subclasses duplicate the functionality of their parent. No classes violate the substitution principle.	Polymorphism is used and some subclasses duplicate the functionality of their parent. No classes violate the substitution principle.	Polymorphism is used but subclasses duplicate some functionality of their parent or violate the substitution principle.	Polymorphism has not been used.
Contract Programming (10%)	Where appropriate, classes include documented class invariants and pre/post-conditions on public members.	Where appropriate, most classes include documented class invariants and pre/post-conditions on public members.	Some classes have documented their class invariants and pre/post-conditions on public members.	Some classes lack important documentation of their class invariants and/or pre/post-conditions on public members.	Minimal or no attempt has been made to document the invariants/pre/post-conditions on public members.

Code Style The Code Style category is marked starting with a mark of 10. Every occurrence of a style violation in your solution, as detected by *Checkstyle* using the course-provided configuration[4], 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 8. Note that multiple style violations of the same type will each result in a 1 mark deduction.

$$S = max(0, 10 - \text{Number of style violations})$$

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. OpenJDK 21 with the JUnit 4 library 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 at least the following directories:

Implementation Code Files:

`src/farm/core/farmgrid`
- All classes you created/modified in the refactor.

`src/farm/core/FarmManager.java`
- Modified `FarmManager`, only if changed from provided.

`src/farm/files`
- All classes defined in files package. At a minimum includes: `FileLoader.java` & `FileSaver.java`

`src/farm/debugged/farmgrid/FarmGrid.java`
- Debugged `FarmGrid` before refactoring.

[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.

JUnit Test Files:

```
test/farm/inventory/BasicInventoryTest.java
test/farm/inventory/FancyInventoryTest.java
```

Do not include the provided code outside of these packages. If you create additional packages, include them in the submission. Only submit code you have modified or created.

Ensure that your classes and interfaces correctly declare the package they are within. For example, `FarmGrid.java` should declare `package farm.core.farmgrid;`.

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.

Provided tests A small number of the unit tests used for assessing Functionality (F) are provided in Gradescope, which can be used 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 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 it's 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`

Change Log

Revision: 1.0.0

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

Appendix

A JUnit Test Marking

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

> **correct** implementation of the respective class
> e.g. `BasicInventory.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. `BasicInventory`Test.java) against the correct implementation of the respective classes (e.g. `BasicInventory.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 `BasicInventoryTest.java` and 4 unit tests in `FancyInventoryTest.java`, and they all pass (i.e. none result in `Assert.fail()` in JUnit4).

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

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

`FancyInventoryTest.java` should still pass 4 unit tests.
However, we would expect that `BasicInventoryTest.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 `BasicInventory.java`.

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