PROGRAMMING IN THE LARGE (CSSE2002)

ASSIGNMENT 1 — SEMESTER 2, 2024

SCHOOL OF EECS

THE UNIVERSITY OF QUEENSLAND

Due August 27th 13:00 AEST

One must learn by doing the thing; for though you think
you know it, you have no certainty, until you try.
— Sophocles

DO NOT DISTRIBUTE. Version 1.1.1

Overview This assignment delivers practical experience developing a Java project based on a supplied specification. The specification is provided in the form of JavaDocs, which describe the classes and interfaces that your assignment must implement.

You will be assessed on your ability to;

- implement a program that complies with the specification,
- and develop code that conforms to the style conventions of the course.

Task The CSSE2002 teaching team want to create a new text-based farming game, but they are so busy with teaching that they need your help to implement their vision! In this prototype first version of the game, Farmer Ali is opening a small store to sell the goods she has made on her farm. She is a very organised farmer, so keeps track of her shop's stock, as well as the customers and sales records.[1]

The teaching team has developed the specification and have already implemented some parts of it, but need you to complete the project following the specification.

[1] In Assignment 2, they plan to take your implementation and expand the game further.

Common Mistakes Please carefully read Appendix A. It outlines common and critical mistakes which you must avoid to prevent a loss of marks. 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. Code supplied by course staff (from this semester) is acceptable, but must be clearly acknowledged. Code generated by third-party tools is also acceptable, but must also be clearly acknowledged, see *Generative Artificial Intelligence* below. You must be familiar with the school policy on plagiarism:

`https://uq.mu/rl553`

If you have questions about what is acceptable, please ask course staff.

Generative Artificial Intelligence You are strongly encouraged to *not* use 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, by hand, in the final exam. If you do use AI tools, you must clearly acknowledge this in your submission. See Appendix B for details on how to acknowledge the use of generative AI tools. Even if acknowledged, you will need to be able to explain any code you submit.

Interviews In order to maintain assessment integrity and in accordance with 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.

Specification

The specification document is provided in the form of JavaDocs.

- Implement the classes and interfaces *exactly* as described in the JavaDocs.
- 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 and *protected* methods as you see fit.

To view the Javadoc specification, please open the link on Blackboard under the `Assessment -> Assignment 1` section.

Getting Started

To get started, download the provided code from Blackboard. This archive includes code for the user interface components, and some unit tests for Stage 0 code. Extract the archive in a directory and open it with IntelliJ.

Tasks

1. Fully implement each of the classes and interfaces described in the Javadoc specification.

For this assignment assume that **nothing** will be null, thus they don't need to be accounted for.

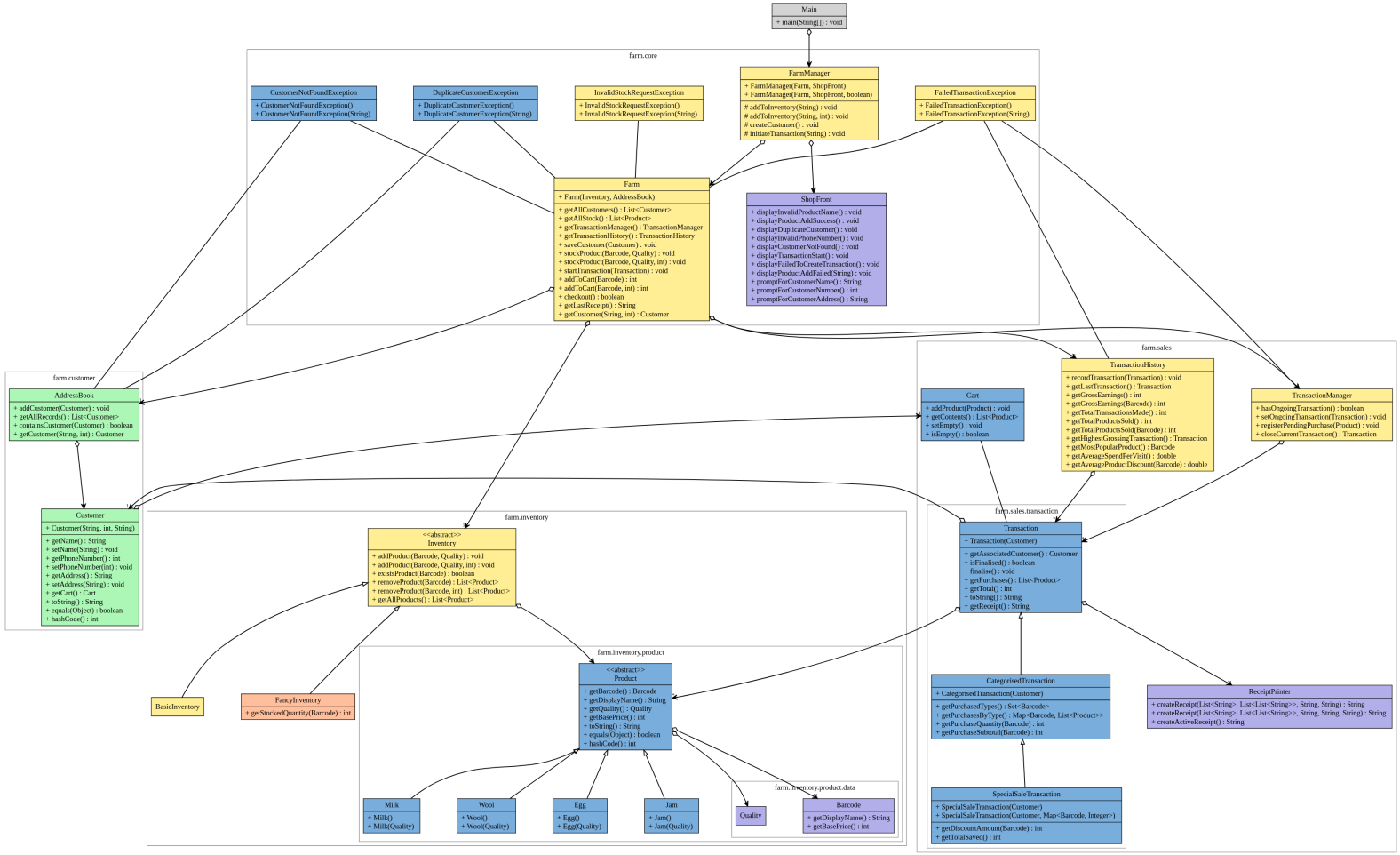

Figure 1: Class diagram of the specification for assignment 1.

Project Overview

farm.core This package contains the interface `FarmManager` between the model of the farm shop, `Farm`, and the provided user interface `ShopFront`. It also handles the details of custom exceptions.

farm.customer This package contains implementations of the `AddressBook` and `Customer` classes. It allows storage and retrieval of customer information.

If an existing customer is added to the AddressBook, a `DuplicateCustomerException` is thrown, and if a customer cannot be found, a `CustomerNotFoundException` is thrown.

farm.sales This package stores implementations relevant to processing of new sales, `TransactionManager` and `cart`, and retrieval of old sale information, `TransactionHistory`.

This package also throws relevant exceptions, and contains the provided `ReceiptPrinter` class used by the UI.

farm.sales.transaction This package stores information relevant to different transaction types, used depending whether products should be grouped or are able to be discounted.

farm.inventory Contains implementations of the abstract `Inventory` class and its children, used to track the products stored by the farm ready to sell.

farm.inventory.product This package contains implementations of the different products which can be stored in the inventory, including information about quality.

farm.inventory.product.data Provided implementation of product information, and quality, enums.

Stages

The assignment is decomposed into stages to encourage incremental development. It is recommended that you complete each stage before moving on to the next. The provided `Main` class allows you to run each stage individually by uncommenting the appropriate lines in the `main` method. Figure 1 highlights the classes that you will implement in each stage: green for Stage 0, blue for Stage 1, yellow for Stage 2, red for Stage 3, and purple for provided code. At each stage, ensure that you thoroughly test your implementation.

Some classes are broken up across multiple stages, this will be documented in the Javadoc, if you are following along with the stages you should only implement the methods for that stage.

Stage 0 Create a simple implementation of the farmer's `AddressBook`, which stores `Customer`. Both classes should be within the `farm.customer` package. A small number of JUnit tests are provided for you to check your implementation. After implementing the `AddressBook` and `Customer` classes, you should ensure you submit to Gradescope to check your submission meets specifications of submission and the autograder is able to run.[a]

[a] You are encouraged to perform your own testing throughout all stages of this assignment, in addition to any Gradescope or provided tests

Stage 1 Implement the `Product` class and its subclasses, the `Transaction` class and its subclasses, and the `Cart` class. This will provide the foundation allowing the higher-level interactions to take place within the program. Also, revisit the classes from Stage 0 and implement their stage 1 features including equality, no duplicate customers, and exception handling.

Stage 2 Complete the interactions between classes by implementing the `TransactionManager`, `TransactionHistory`, and `BasicInventory` classes, as well as implementing the `Farm`.

Stage 3 Implement the `FancyInventory` class in order to extend the capabilities of the program usage and to support quantities.

Usage

Within the command line interface, each area of the shop has a series of commands which can be used.

Table 1: **CLI initial usage commands**

Command	*Description*
q	Quit the application.
inventory	Manage the farm's inventory.
address	Manage the farm's address book.
sales	Enter the sales mode.
history	View the farm's sales history.

Table 2: **CLI inventory mode commands**

Command	*Description*
q	Quit the inventory mode.
add <product-name>[quantity]	Add a product to the inventory, with a given quantity if Fancy.
add -o	List all the product type options available to be stocked.
list	List all the products currently stocked in the inventory.

Table 3: **CLI Address book mode commands**

Command	*Description*
q	Quit the address book mode.
add	Add a customer to the address book.
list	List all customers in the address book.

Table 4: **CLI Sales mode commands**

Command	*Description*
q	Quit the sales mode.
start [-specialsale |-categorised]	Begin processing a new transaction. [Optional transaction type].
add <product-name>[quantity]	Place the specified product in the current customer's cart, with a given quantity Note: There must already be an ongoing transaction.
add -o	List all the product type options available to be sold.
checkout	Finalise the sale of the products in the current customer's cart.

Table 5: **CLI Sales history mode commands**

Command	*Description*
q	Quit the sales history mode.
stats [<product-name>]	Get the total stats for the shop. [Optional product stats]
last	Prints the receipt of the last transaction made.
grossing	Prints the receipt of the highest grossing transaction.
popular	Displays the name of the most sold product.

Marking

The assignment will be marked out of 100. The marks will be divided into two categories: functionality (F), and style (S).[2]

	Weight	*Description*
F	90	The program is functional with respect to the specification.
S	10	Code style conforms to course style guides.

The overall assignment mark is defined as

$$A_1 = (90 \times F) + (10 \times S)$$

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.

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[3], 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.

$$S = \frac{max(0, 10 - v)}{10}$$

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 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**.

[2] While not assessed on writing tests for this code, you are strongly encouraged to develop your own tests.

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

Submission

You should submit all files that you are required to create or alter, but not those that have been provided to you in full.
Submission is via Gradescope. 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 *before* the deadline. Code 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 as many times as you wish before the due date. Your last submission made before the due date will be marked.

What to Submit Your submission should have the following internal structure:

`src/`	folders (packages) and `.java` files for classes described in the Javadoc.
`ai/README.txt`	any file(s) you have within the ai folder - see Appendix B for details

Included in the provided code are `bundle_unix.sh`, `bundle_windows.bat`, and `chronical.ps1` files.
- For MacOS and Unix users, double-click the `unix_bundle.sh` file to execute it.
- For Windows users, double-click the `windows_bundle.bat` file to execute it. Unfortunately due to the limitations of Windows environment a zip file is not created. Instead you will need to manually zip the contents, `src` folder, inside the generated `zip_my_contents` file to create ur `.zip` file.

A complete submission at the end of Stage 3 would look like:

```
src/farm/core/Farm.java
src/farm/core/FarmManager.java
src/farm/core/CustomerNotFoundException.java
src/farm/core/DuplicateCustomerException.java
src/farm/core/FailedTransactionException.java
src/farm/core/InvalidStockRequestException.java

src/farm/customer/AddressBook.java
src/farm/customer/Customer.java

src/farm/inventory/Inventory.java
src/farm/inventory/BasicInventory.java
src/farm/inventory/FancyInventory.java

src/farm/inventory/product/Product.java
src/farm/inventory/product/Egg.java
src/farm/inventory/product/Jam.java
src/farm/inventory/product/Milk.java
src/farm/inventory/product/Wool.java

src/farm/sales/Cart.java
src/farm/sales/TransactionManager.java
src/farm/sales/TransactionHistory.java

src/farm/sales/transaction/Transaction.java
src/farm/sales/transaction/CategorisedTransaction.java
src/farm/sales/transaction/SpecialSaleTransaction.java

ai/README.txt
ai/*
```

Ensure that your classes and interfaces correctly declare the package they are within. For example, `Egg.java` should declare `package farm.inventory.product;`.

Do not submit any other files (e.g. no `.class` files).

Provided tests A proportion 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 visible unit tests on Gradescope does *not* guarantee that you will pass all the tests used for functionality marking.

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 13:00 but finishes after 13:00 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:

`https://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:

`https://uq.mu/rl552`

Change Log *Version: 1.1.1*

v1.1.1

Provided Code `FarmManager` has been updated to remove known bugs in its implementation. You are encouraged to update your `FarmManager`, however using the old version will not affect your implementation or final score.

- Fixed known bug from v1.1.0 in `FarmManager.handleHistoryStats()`, to actually display the stats amounts in dollars not cents.
- Fixed known bug in `FarmManager.launchInventoryMode()`, that caused game to crash if you listed an empty inventory.

Task Sheet

- Removed conflicting statement about making protected methods. You are **allowed** to make protected methods and constructors. Just note that protected attributes violate the style checker.

JavaDocs

- Clarify what to do if the product name is invalid in `FarmManager.addToInventory()`.
- Added stage tag to `CategorisedTransaction`.

v1.1.0

Provided Code

- Remove erroneous print statement from `ReceiptPrinter.buildPurchasesSectionEntries()` line 161.

JavaDocs

- Correct heading from `"Price (ea)"` to `"Price (ea.)"` in description of `CategorisedTransaction.getReceipt()`.
- Fix usage examples of `SpecialSaleTransaction.getReceipt()` to pass in the `discounts` map as an argument.
- Clarify expected format of discounts in `SpecialSaleTransaction.toString()`.
- Clarify expected format of prices as cents in `TransactionHistory` methods that return price values.
- Clarify definition of total income in `TransactionHistory.getGrossEarnings()` and `TransactionHistory.getGrossEarnings(Barcode)`.

Announcements & Known Issues

- Non-Temurin JDK appears to cause build problems with JUnit. See EdStem or a tutor for resolution.
- Examples in the `SpecialSaleTransaction.toString()` suggest that it should begin with `"Transaction"` rather than `"SpecialSaleTransaction"`, and the associated JavaDoc does not specify anything suggesting otherwise. Standard practice is generally that the simple class name should be used here, so that `CategorisedTransaction.toString()` and `SpecialSaleTransaction.toString()` would begin with `"CategorisedTransaction"` and `"SpecialSaleTransaction"` respectively, but in order to avoid major changes to the previously specified behaviour, the original requirement that all subclasses still begin only with "Transaction" will be retained.
- The provided code contains a known bug in `FarmManager.handleHistoryStats()`, which incorrectly outputs prices as dollar amounts when they are actually cents. This will not be patched. It will not affect your implementation (or our testing) of `TransactionHistory` methods, which you should continue to implement as specified.

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 data members 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 members and protected methods 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 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 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.

