Seng 201 Farming Simulator Project Report

Manjeet Panwar – msp87
James Harris – jeh128

Structure and Design Choices

The most integral design decision made during this project was to, wherever practicable, make expandability as easy as possible.

In the pursuit of expandability, the decision was made that the maximum decomposition of classes for purchasable things (Animals, Crops, Items) would be taken. This was achieved by using inheritance. This can be seen on the supplied UML Class diagram. It allows new purchasable things to be added easily, or for existing ones to be updated. Individual items inherit item type super classes which intern inherit from the Item superclass. Making each item a grandchild of the item class. This allows all the items the user owns to be kept in an array-list of type Item. It also allows each item type to have different parameters. Crops and Animals directly inherit from their respective super classes, as there is not need for further subdivision or grouping.

The individual purchasable thing classes themselves are very simple, essentially just acting as constructors for their super classes, but with two static fields. However, they lend themselves to easily be expanded. For example, if the ability to milk cows was to be added. It is much easier with the current class hierarchy, that with for example, if all animals were just instances of the animal class.

The decision to have static purchase prices and descriptions for each purchasable thing was made partway into the implementation of the project. This was so when the shop screen is being built, it did not need to instantiate objects of each type to access their prices and descriptions, just to then throw them away. This was done by creating an interface Purchasable which all purchasable things implement. See UML Diagram. This interface guarantees that purchasable things have both a purchase price and a description.
This same problem was solved in the Command-Line implementation by adding a dictionary of prices. However, it was determined that this solution did not lend itself to easy expandability. So was improved when making the GUI.

All the state information of the application is stored in the gameState class. All logical processing and gameState updating is handled by the GUIGameEnviroment class. The GUI itself it implemented by the FarmingSimulatorApp class. Which interacts primarily with the gameEnviroment. Its only interacts with gameState when creating a new gameState from user inputs. The gameState holds a Farm object which is where the bulk of game information is stored.

Encapsulation is used throughout the project to protect variables. All variables are private and may only be accessed via getters and setters, or other methods designed to specifically change their values in a certain way. This means all communication between classes is via method calls. This gives each class control over how its variables are changed. In most cases other methods are preferred to setters. For example, an increment method that increases the value of a variable by one. Instead of having another class requesting the change first call for the current value from a getter, then using a setter to set the value to be one higher than it was.

Unit Testing

Unit tests were used for all simple classes, and a selection of methods from more advanced classes. Many of these tests are checking simple getters/setters or basic change methods. However, some tests cover more advanced method. As can be seen in Figure 1, a code coverage of 100% was achieved on all simple classes. However, 100% code coverage does not ensure that code will work correctly. To test methods properly, the blue-skies, boundary, and invalid approach was taken. In which tests are designed to cover all equivalence sets. See Figure 2 for an equivalence partition drawn for GUIGameEnviroments checkLegalName method.

The code coverage for the entire project is inadequately low, as it was stated in the specification to only write unit tests for the simpler classes. It would also be difficult to test for user inputs.

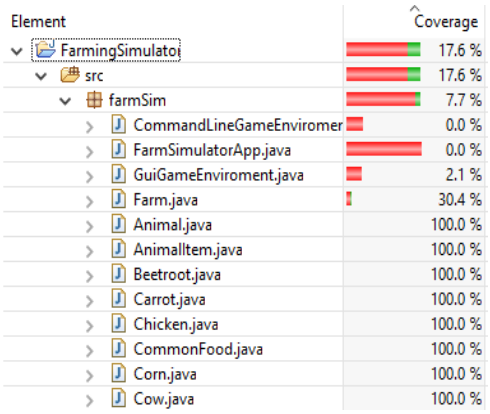

Figure 2. Code Coverage Report For Unit Tests

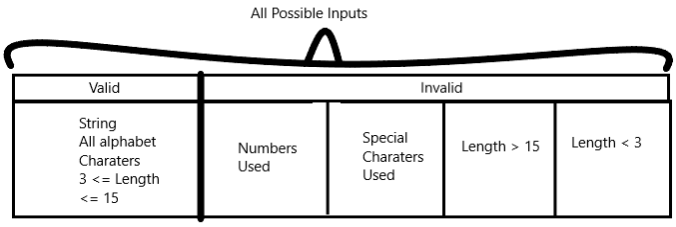

Figure 1. Equivalence Partition Example

Thoughts on the project

The project went as well as can be expected. The lockdown had little impact on our ability to achieve our weekly goals. The scope of the project seemed a little large for the time given and grade value. The project easily could have been replaced with a smaller one, while still providing the opportunity to experience the software development process and use objected orientated programming features.

Retrospective

A semi-iterative approach to the design process was applied. In which as problems were discovered during implementation, plans were updated and improved. Then methods were updated or restructured. However, this was not entirely intentional. As problems only arose due to mistakes in the planning. For example, not originally planning to have the Purchasable interface with static prices and descriptions for purchasable things. This meant the in the command-line implementation, a dictionary was used for storing prices. This meant when moving to the GUI implementation, all purchasable subclasses needed to be altered. It was also only after the command-line implementation that the decision was made to decouple gameState from the game environment.

Similarly, game logic was originally coupled to the FarmSimulatorApp class, instead of being handled in its own class. Only after the functionality of the GUI was finished that the logic was decoupled. Which was quite a time sink.

These problems and others could have been avoided by spending more time at the beginning planning how the project would be structured. In future projects more attention will certainly be paid to planning the structure.

Effort Summary

Effort in hours:
James – 55
Manjeet – 25

Statement of agreed % contribution

James - 70%
Manjeet - 30%