Seng201 Project Report

William Huang 37969552
James Napier 79030922

The structure of our application is highly modularised into class objects that may stand alone outside of the game such as the animal and crop classes and their item counterparts. This design enables easy coding as these classes only preserve their own attribute values with little to no modification to other classes. Therefore it is clear during the design and testing phases where a bug is caused because their private attributes are only modified within their own class, apart from public setter methods. However, we limit the call to these methods by mostly invoking them only when necessary and limiting the complexity of these functions to avoid unexpected bugs to occur. For example, item objects for animal and crop do not contain any setter methods because their values are automatically set once instantiated based on what their constructor string name is and do not change. Animal and Crop setter methods on the other hand, have internal constraints in place to make sure values are in the correct range. Within the set health and happiness methods of the animal class, the settable ranges are bounded between 0 and 10. Likewise with Crop, the mature method will not go below 0. We decided to keep all items and crops within the same class and not make a different class for each separate item type. The decision behind this process was that the items and crops were too similar to create individual classes for. This would just cause redundant coding without changing any functionality, so we limited inheritance just within the animal classes.

The Farm class is the main class that ties all other game classes together. This is very efficient as there is only ever 1 instance of a Farm class per game. It is intuitive to make this the main class of the game where all other classes are stored within. It is easy to access every single object when calling functions such as feed(), play() and harvest() as all objects of the farm are stored within their own ArrayList container meaning it only takes 1 function call to just 1 class to carry out these actions. The ArrayLists of the Farm class store the object instances of each class, not just a string representation. This is very important as through the Farm class, we are able to get each item of the array and invoke their own method calls with a simple For-Loop. It also keeps track of the attributes for each individual object instance like the animal's health and happiness or a crop's harvest time. Cohesion between class communication is high due to us sticking to small integer values. Therefore, no conversions need to be done when different classes are acting with one another. For example, animals and crops restore the same amount of stats as an item yields and the same money currency is used between the shop, farm and animal/crop incomes. The Farm class is stored in the Main class to separate GUI functionality from game logic further; reducing the size of Farm and making window flow very obvious when inspecting Main.
Our unit test coverage is 25.9%. At first glance, this may appear pretty poor but upon further inspection, this outcome is a lot more desirable than expected. First off, almost all (~3,800 of the 4,100 = 93%) of the missed instructions were in our GUI classes which are not relevant to unit testing since game functionality does not come from these classes which distort the test coverage figure hugely. However, we do not write off testing GUI classes completely. User testing of the GUIs are done instead which is a more reliable test method and is not accounted for in the test coverage. Focussing our attention on testing of classes with in-game functionality, we will find that our average test coverage is well above 50%. This outcome is more desirable than it may seem as the majority of the missed cases are setter and getter methods which create redundant test cases. Therefore the unit test coverage percentage of 25.9% is very deceiving as we do manage to cover a majority of the functionality of our code and interactions between different classes together with unit testing.

We think the project went very smoothly with all outcomes considered because the scope of the project was quite clear with little to no ambiguity. The fact that random events were made non-compulsory also made the project easier to complete. During planning, it was intuitive which classes would be made as classes based on the project scope. Having a hierarchical structure with a Farm containing all the other classes was instinctive and advantageous as it is elegant in the coding perspective because it prevents references between objects all over the place.

The use of Github and Discord made communication and implementation much easier. This combination allowed easy sharing of code and clear verbal communication allowed us to implement different parts of the project simultaneously with minimal github conflicts.

The scope of the project was also quite intuitive. This allowed us to have similar concepts of implementation without clashes of ideas.

If you have noticed our weekly project effort registry, our work distribution is very heavily distributed towards the due date of the project. In general, this is bad practise but we didn't take initiative to start sooner and we both happen to be procrastinators. Despite this, we are pleased with the end result.

The obvious improvement for the next project is to distribute our time allocation more uniformly like normal human beings. Another improvement is our gameplay functionality. Due to time constraints, we spent most of our time developing the functionality of the game and project requirements and little time on the gameplay. If the goal of the game is to obtain the highest score possible at the end of the game, then it is clear that the dominant strategy is to purchase animals instead of crops as animals yield much more money than crops by a large margin and require less maintenance.

James and William spent 40 hours working on the project and have a contribution percentage of 50% each.