Daniel Chapman 67098951, Tom Rizzi 71582886

Inheritance plays a big role in creating the objects for the farm in our project. The items, animals, crops, and different types of farms all extend respective parent classes. For the farm types, as each farm type only manipulated one attribute of the farm object, and thus only one attribute or method would need to be overwritten, using inheritance made the most sense. As the methods being overrode were the same for three of the four farm types, using an abstract method would require an identical method in three separate classes. Thus, compared to this abstract approach, and an interface class approach, where each different farm type would require essentially the same functions apart from one or two, using inheritance was an easy choice.

We also decided to implement parent classes for each of the animals and crops that the player could purchase. As each animal is functionally the same for our game, mostly only having getter and setter methods, we only needed to change the specific starting attributes for each different object. Hence, creating one parent class for animals, and then creating subclasses for each animal that passed in specific parameters was, again, an obvious choice to us over creating an animal interface. The same applies to the crops; as each crop is functionally the same, we only need to pass in specific parameters when creating different objects through the subclasses.

As specific items were designed for use on animals, while others were for crops, we decided to implement the item class as an abstract class, with an abstract method to get the benefit of the item. We decided to do this as an item for use on a crop has its benefit measured in days, whereas an item for an animal has a benefit that adds a percentage of health. Because they both have a benefit, just as a different type, it made sense to use an abstract method to handle the benefit for each type of item.

Unit test coverage is a measure of the quality and comprehensiveness of unit tests. In our project we used the JUnit API, which measures the unit test coverage as a percentage of instructions covered when the tests are run. The unit test coverage for our project was only 43.4%. While this does seem low, a large percentage of our code is unsuited to unit testing. This is mainly due to our game being implemented using a GUI for user interaction; a large percentage of our code involves constructing, launching, and closing different GUI screens, which presents unique challenges with testing. Most of the GUI's functionality is interacted with using event handlers, such as when a button is clicked. This makes testing these classes difficult, as we would have to simulate clicking on different objects with different inputs. Instead of doing this, we manually tested the functionality of the GUI, by bringing in third-party testers. This allowed us to observe how an average user may interact with the system, and thus, allowed us to see where major bugs may occur during normal operation of the system. By playing the game ourselves, we could also perform white-box testing, ensuring that exceptions were thrown when they were needed to, and that the functionality we included to catch exceptional behaviour caught, prevented, and dealt with the problems as they occurred.

In retrospect, there were many aspects of the development processes that worked effectively and others that were a hinderance. We used a distributed Git repository to enable us to collaborate on the project. This allowed us to work independently on the project without worrying about overriding each other's work. This was, however, the first time we had used Git on a collaborative project, and as such, there were times when we had developmental delays due to using Git's features. On multiple occasions, if we worked on the same file, or were working at the same time, and one of us tried to push their work to the repository, it ended up overriding the changes that the other had committed, due to mistakes made during the merge. This led to us being delayed for a significant amount of time while trying to re-do the overwritten changes. A better understanding of Git and its operation will be particularly useful in future collaborative projects, as we will know how to deal with issues like this in a timelier manner, or even better, avoid them in the first place.

Another aspect of our development process that worked well was the use of an integrated development environment. This allowed us to efficiently write our program, as the syntax of our code was constantly being checked. It also allowed us to quickly identify where errors were, quickly navigate to them, and identify what the problem was and how to fix it. There were a few times where using an IDE was a hinderance though; due to unfamiliarity, we lost side windows that aided us with development, and thus, had to pause development to locate and reopen them.

As neither of us had familiarity with the Java language before taking SENG201, we were learning features as we created the game. Multiple times throughout this project, we had to refer to the lecture notes or the API online to determine what methods or tools would be appropriate for what we were trying to accomplish with a specific class. While this did not hinder us to the point where we were majorly set back, it did slow us down slightly. However, now that we have a better understanding of the Java language, any further projects we work on will benefit from the learning experience.

As shown in our initial UML class diagram, we at first thought we would be able to encapsulate all the game logic into one large class. It was not until we had upwards of 700 lines of code in one overly large class that we realised how inefficient and complicated one main logic class can be. Thus, a significant amount of time was spent reformatting and reworking the logic of the game into several packages, and then into several different logic classes, to make our code tidier and more readable. This is an improvement we will take with us into future projects; interaction between multiple classes is far more readable and accessible than one 'god' class.

Effort Distribution

Tom – 40% - 28 Hours

Dan – 60% - 43 Hours