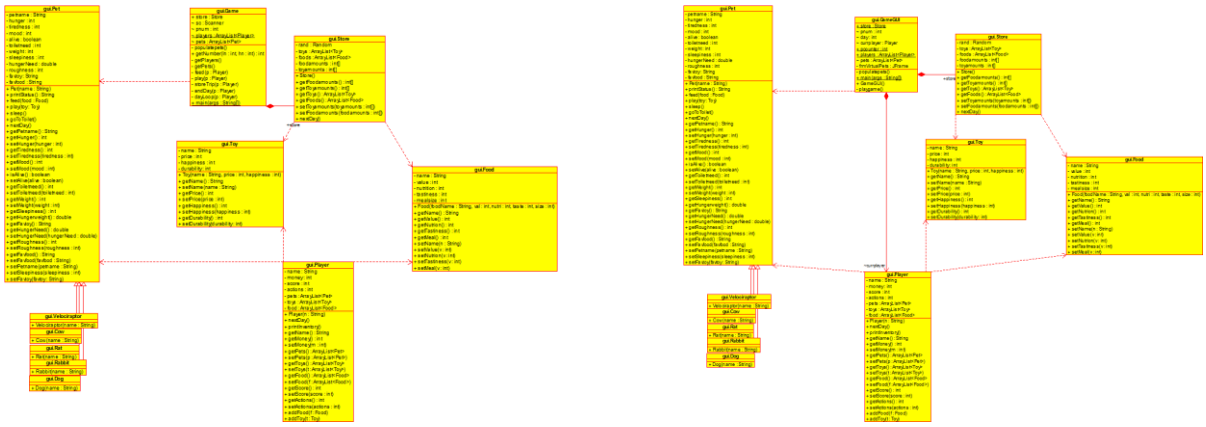
Matthew Fagan – 15700612 Luke Woollett 47870172

SENG201 Virtual Pets Assignment



The main class for each of the game variants (Command Line and GUI) is the Game class. This class is what runs the game, creating the displays and calling on each of the other classes when they are needed. The game class starts off by creating a given number of Player classes, dictated by the user. It uses a scanner to collect the names of the players and assigns them. After the Players are created and stored in an array the Pets are created and stored in arrays for each of the Players.

After collecting the Players and all the Pets for the Player the game iterates over each day for each Player. The Players are given 2 actions to use on each day and use those on their Pets or to buy Food or Toys. If the player chooses to buy Food or Toys the Game class calls upon the Store class to return the Toys/Food it has and the number of stock it has of each item, it then reduces the number of the bought item by 1 when bought. The Game class then puts the Toy/Food into the player's inventory.

Once the number of actions the player has left reaches 0 the Game class then iterates onto the next player. Once all players have used their actions for the day the Game uses the nextDay method. This method iterates over all the Pets using their nextDay method which reduces their stats accordingly with their subclass attributes, uses the nextDay method for the Store, which increases the stock in the Store by a random amount each night.

The only class that uses inheritance is the Pet class, with each of the individual pets being a subclass. None of them were given extra methods, but each of them were initialised with their own individual base stats that are unique to each subclass.

Some of the design choices that we had to make were the class inheritance structure and the actual user interfaces. The class structure can be seen in the UML diagrams above with the command line UML on the left, and the GUI on the right. The only inheritance used was in the Pet class, to create all the pet animals and their attributes. This was done to keep all the animals separate, and their own being. A couple of design choices that were made were: Not implementing the reviving of pets, as we felt that it was stupid and unrealistic, and not adding in the Preview of pets stats. The latter was not added in, as when reading over the requirements in the last day, we saw that titbit of information, which would've required a whole rebuild of a window that was already spaghetti code.

The classes that were tested were the Player and Pet classes. These two classes have quite a few methods with some boundary cases that could break the code. The methods in these classes that were tested using Junit tests were the methods that aren't getters or setters. This resulted in a low

coverage because not every single method was tested for correctness, and not every method tested was given all possible boundary values (or even a range of expected values.) The assignment isn't too hard; the code looked reasonably easy but the scope of the project is huge. When designing the program, it was hard to think about what the markers were going to look for and to code that into the solution to show that we are capable. My (Luke) thoughts on the assignment was that, as Matthew said, that the code was easy, but getting your head around how GUI elements interacted and worked with each other was the real hard part.

Some of the things that went well during this project were the communication between the partners, we both knew what was going on and happening all the time. Another thing that went well was the UML design at the start of the project. The classes were all thought out before the project was started, with all the methods and attributes accounted for before the coding started. This made it easier to focus on how to code, instead of what to code.

What didn't go so well was the boundary case testing along the way. One bug that we found out about right at the end of the code was that if there was more than one of a specific kind of pet they would all have the same name. This would have been easy to fix if boundary cases had been tested as we went.

The improvements that could be made for next time would be more planning, and more testing as the code progresses, possibly even some JUnit tests written before the code starts to get written. Planning the code before the coding begins was something we thought that we should do, and it turned out to be one of the biggest helps. Planning meant that we knew exactly what we were supposed to be coding, without having to guess what classes to create as we went. More testing along the way would have been beneficial as we wouldn't have gotten right to the end of the project, done a few tests and then realised we made a mistake right at the start that would have been an easy fix at the time, but is impossible to fix without going back and re writing the entire code.

Contributions:

Luke Woollett – 50%
Matthew Fagan – 50%

Luke's Contribution: All the code, JavaDoc, some unit tests for classes that weren't covered already.

Matthew's Contribution: The report, Unit Tests, Code design, UML diagrams.