SENG201 Island Trader Project Report

Frederik Markwell (fma107) - 51118501
Andrew Hall (amh284) - 98337995

Structure:
Our application is structured with an architecture similar to the three tier architecture discussed in the lectures. The user interacts with the game through a UI class (CommandLineInterface, or the various GUI classes). As much as possible, these classes contain only the code to take input from the user and return output to them.

The logic of the game (the “application layer”) is contained in GameEnvironment. A GameEnvironment object is created when a new game is started. The command line and GUI classes call methods on this object to process player actions and to get the resulting data that needs to be returned to the user.

Another aspect of the application layer was the game’s “state”. When the game is first booted, the environment is put into a “SETUP” state. When the player picks a route to travel, the environment is put into a “SAILING” state. If the player gets boarded without cargo, the environment is put into a “GAME_OVER” state, and so on. This design choice allows us to “flatten” the logic of the application so we didn’t need to handle all the possible choices the user could make - each action just changes the state, and then the UI class opens the next view based on that.

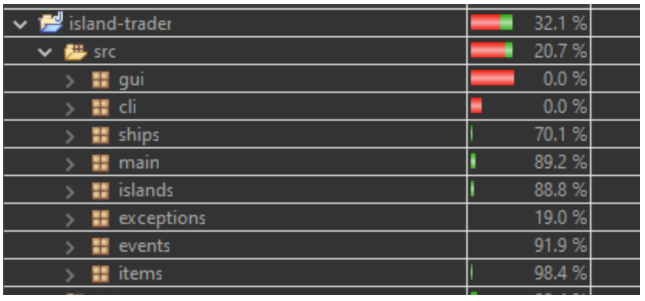
Finally, the remaining classes make up the data layer. Different objects are nested inside one another to create the storage structure of the world. For example, an Island object contains a Store object, which in turn has a list of Item objects. We also use inheritance and abstract classes extensively to represent different types of the same object. For instance, TallShip, WarShip, Clipper, and Barge all inherit from the abstract class Ship.

It is important to note that our project does not precisely conform to the three tier architecture because we have interactions between non-adjacent layers. The command line and GUI classes get objects from the environment class and call methods on them, both to get data and to update them. We made this decision to allow us to better break down the game logic into smaller functional units that are then easier to manage, and avoid creating a so-called “god-class”.

Code coverage:

Our overall test coverage was quite low, around 32%. However, when broken down by category, the results look a lot better. Coverage for the actual code of the game logic is between 80 and 100%.

This discrepancy is because we only created tests for things that contained complex logic, and therefore needed testing, like GameEnvironment and the data classes (Ship, Item, etc). UI classes have fairly simple logic and were tested manually.



Thoughts and Feedback
The project brief was clear and easy to understand, we knew what we had to do. The idea for the type of game was good, and having played similar games in the past we had some idea of how we wanted the finished product to end up.

The scope of the assignment aligned well with the time allotted, but there were some things that we learned in later SENG201 lectures, such as design patterns that would have been good to know before starting the project. We also found some of the requirements in the brief, such as each island having to have a route to each other island, somewhat limiting.

Retrospective
On the whole, the project turned out well. We met up pretty early, communicated often, and were on the same page with most issues related to the project. We planned out the structure and flow of our game together in our first few meetings, and the work was divided pretty evenly. Before we started coding, we set up a repository on GitLab, and then used Git effectively to collaborate throughout the process.

There were some areas in the process where we could have improved a little bit. Initially, the plan was to space out development over the entire period we had to work on the project, leaving at least a week at the end. Our estimates were slightly optimistic, and other university work meant that we had to work significantly harder in the final few weeks of the project to get it in on time. On the other hand though, we did leave enough time that with that extra effort we were not in danger of missing the deadline.

Another area was that we originally used waterfall development. The first few weeks were spent creating and designing classes, without knowing how they'd fit into the game beyond our initial UML diagrams, or even having an executable. This caused early development to lack direction, as we created features without being able to see how well they work.

For our next project, we would use a more agile development process from the start. Iterating over a working product would help us to see what is really necessary, and also to test that our code integrates earlier in the process.

Effort Spent
Frederik - 45 hours
Andrew - 40 hours

Note that these numbers are estimations, and may not match the data of the weekly progress reports.

Contribution distribution:

We both agree that the contribution was split 60-40, that is, Frederik contributed 60%, and Andrew contributed 40% to the final product.