SENG201 Report

Sean Reitsma – 61687985

Caleb Gourley - 12407939

**Design decisions**

We began design for the project by sketching first some basic use case diagrams and then the class diagram. Because setting up the game and playing the game are distinct sets of features, we created one use case diagram for each to make them simpler and easier to understand. We then drafted a class diagram. The design of this project was largely bottom up, that is, first we designed the model classes to represent entities and data, then we designed the service classes, and then we designed the gui and its classes. The main model classes are Tower, Cart, Upgrade, Player, Inventory, and Round. We also created the interface Purchasable to represent items in the shop. In addition, we made repeated use of enum types to represent groups of constant values, such as the Rarity, ResourceType, and Difficulty enums. Upgrade started out as a very basic class which severly limited the scope of what upgrades could do. Now Upgrade is a generic abstract class which all upgrades inherit from, and all upgrades must implement the applyUpgrade method. This provides a unified interface for applying and interacting with Upgrades but allowing wider scope for what they can do and making it easy to add more. The service classes consist of RoundService, ShopService, and SetupService. These classes all provide useful methods which the controller classes use to manipulate the game state. There are also 3 classes which provide javafx Service implementations to handle concurrent movement of Carts and concurrent filling of Carts by Towers. Initially concurrency was handled by java’s built in Executors but this caused issues with certain methods being executed outside the javafx application thread. The RoundDiffPopupController and CellFactory classes have been omitted from the class diagram for simplicity. The RoundDifficulty class is in the models package but should probably really be in the services package as it is only used by RoundService. Round difficulties are generated somewhat randomly by generating random numbers within a range which increases as the round number increases. In order to come up with sane values for each of the rounds we figured out the mathmatical relationship between the track distance required to win a round and the cart size, speed, and number of carts, and the tower size and speed. We then used the round\_difficulty.ods spreadsheet included in the submission to experiment with values and ensure a reasonable difficulty progression through the rounds.

**Test coverage**

For JavaFX we managed the testing in an excel spreadsheet manually to ensure we wouldn’t have bugs and scene builder related issues which covered features that we had implemented in gui. For unit test coverage we tested the core components for model classes and service classes. Certain methods in the service classes, such as the RoundService.playRound() method, couldn’t be unit tested as they relied on javafx concurrency solutions, or were simply to complex to create reliable unit test cases for. We handled this by manually carrying out acceptance testing to ensure that the functionality of such methods was still as expected. This is why we have a high test coverage percent for most of the model classes but a relatively low coverage for the service classes. Of course we have zero unit test coverage for gui classes but that is to be expected.

**Thoughts and feedback**

We setup a Trello board initially but didn’t really use it. Towards the end of the project we instead mostly worked in person and relied on communication heavily when working remotely to know who is working on what and using branches also made it easier to avoid merge conflicts.

We setup our Timeline for the project using a Gantt chart which provided a good outline for us to follow and we stuck to that plan for the first 4 weeks but in that final week the time we had set out to complete javafx classes and service classes we underestimated the amount of time these classes would take which forced our hand and made us have to push back some of our tasks and crunch to get these features implemented which limited our testing coverage to find bugs. It also didn’t help that we didn’t go back to our Gantt and adjust our goals and hours after we initially set it up. Because we didn’t refer back to our plan and adjust it we thought we were on track throughout the project when we were in fact falling behind which just contributed to the extreme increase in workload in the last week.

We did a lot of our work in pair programming (which is partly why most of the commits are Sean’s), and this was when we made the most progress. We worked well problem solving together and sharing ideas in person rather than remotely.

Our class diagrams changed significantly since we first created them in the planning requirements phase. The model classes stayed largely the same, however, with the most significant changes being in the service layer and its helper classes. This is a sign that we were able to adapt our design. We particularly had to adapt to the threading issues we had which required an overhaul of the tasks classes and the logic in roundService that managed them. This was difficult because we haven’t had significant experience in large multithreaded applications.

We should have had a higher static amount of work put into the project each week rather than slowly ramping up our workload this would have left us with more time to deal with risks, issues, and bugs and left our last week to debugging, documentation and packaging up the project.

Improvements that we could make for the next project could include a better planning system and a consistent review of the plan. Something like an agile methodology could have been helpful in this case.

**Hours spent**

Sean – 78

Caleb -

**Percentage contributed**

Sean - 60%

Caleb – 40%

NOTES:

