SENG201 Project Report

Corey Hines (77926357)
William Brown (18751753)

Structure of the Application

When structuring our application, we prioritised using object oriented programming principles such as encapsulation and modularisation. We restricted the visibility of class members as much as possible, and used public getters and setters when necessary.

At the centre of the application is the GameEnvironment. We decided to make this class a singleton, allowing us to get an instance of the GameEnvironment and any of its members anywhere in the code. This alleviated the need of passing many objects through multiple constructors, leading to simpler and cleaner code. For instance, getting an instance of the shop is as simple as calling .getGameEnvrionment.getShop().

Decoupling GUI code with JavaFX Properties

The use of JavaFX properties proved to be particularly useful in decoupling GUI code from main application code. For example, when the money value in the shop changes (when an Item is bought or sold), rather than directly calling into JavaFX code to update the GUI from non-GUI code, we could instead add an event listener hook in the GUI controller class like so:
getGameEnvironment().getShop().getMoneyProperty().addListener(($, oldValue, newValue) -> this.updateGUI(newValue.intValue()));
This made it possible to implement more unit tests, and further applied the principle of keeping a class's functionality self contained.

JavaFX properties additionally allowed us to simplify the callbacks needed for state changes. When the game state is changed, getGameEnvironment().getStateHandler().setState(GameState.ROUND_ACTIVE), different classes will need to behave differently. For example, when the round is paused or active: carts will need to pause/resume themselves, and towers will need to stop/start generating. Using event listeners allowed us to keep things modular and to keep functionality encapsulated/self-contained.

To decouple GUI code from backend code, we added certain classes to handle GUI related interactions. For example, GameMap contains simple backend code for things such as storing the tilemap, placing towers, etc. GUI interactions such as moving towers with the mouse and buying/purchasing items from the shop were implemented in controller classes: GameController or its "sub-controllers": MapInteractionController, InventoryController, ShopController. These GUI controller classes call into the backend code. We made it a point that GUI controller classes contain only GUI related functionality which allows the game logic to be independent and usable/testable by itself.

Items

We have two types of purchasable items in our game, Towers and UpgradeItems.

SENG201 Project Report

The Purchasable interface is used to represent an item purchasable from the shop. It contains constant information such as name, cost price, sell back price, and contains a .create() method to instantiate an instance of the item. The TowerType class implements Purchasable, as we needed to add additional constant properties such as tower reload speed. The Item interface is used to represent instances of items. It provides the method .getPurchasableType() to return the Purchasable representation of the item.
Upgrade items, which only require a single instance of each item (unlike towers), implement both the Purchasable and Item interfaces.

Unit Test Coverage
The overall instruction coverage of our project came to 22%. Excluding the gui package and the game.assets package, this brings our instruction coverage to 51%. We justify this percentage as we opted to not implement test cases for basic getters and setters, or basic functionality that would be caught in regular playtesting. Unit testing of JavaFX functionality is out of the scope of this project.

Thoughts and Feedback
Both of us had some prior experience in either Java or similar object oriented programming languages. However we had limited experience in collaborating with other people on a software project, and using git. As such, developing the game was not as much of a struggle as say perhaps other teams with less experience might have faced. But that does not mean to say we did not face our own challenges, particularly when merging conflicting code when simultaneously working on the code base.

Starting the project early during the term break was very beneficial. Progress was quickly made on the project, with an application somewhat resembling the final product in only a month or so. While there were still some difficulties in merging conflicting code, and some crunching to meet the deadline, overall things went very smoothly. We gained valuable experience in working on a software project in a collaborative manner.

An improvement for further projects would be to communicate which features are delegated to which team member. Some features, particularly in the later stages of the project, were often only communicated after they had been completed. This was not a problem in our development (likely due to chance), however this may become an issue particularly in larger teams, as multiple team members may inadvertently work on the same feature.

Effort and Contribution
William and Corey both spent on average about 10 hours per week working on the project. Including work done during the term break, this comes to a total of about ~80 hours each. The contribution percentage is 50% each.