Tick Attack Final Design Report

Finn Lidbetter, Patrick Coyle

December 6, 2016

1 Introduction

This report details the design decisions that went into creating our final implementation of the Tick Attack game. This document is split up into 5 parts. Section 2 details how the code was designed to adhere to MVC and how it still respects the MVC architectural design pattern. Section 3 discusses the formal design patterns that were used in the implementation. This includes a discussion of some of the design patterns that we considered but ultimately decided not to use. Section 4 describes some of the other design choices that were made that don't really fit into the discussions about MVC and the design patterns. Section 5 details the design decisions made specifically for milestone 5, including a description of elements of previous iterations of the Tick Attack implementation that were adapted in order to facilitate the required additions. Finally, section 6 describes the testing process for our game.

2 MVC Adherence

Our submission for the Tick Attack project follows a Model/View/Controller architectural design pattern. That is, the code is designed such that the parts of the Model only interacts with parts of the Controller and the View also only interacts with parts of the Controller. The View is made up by the following classes and interfaces:

- SimpleViewer
- IView (interface)

The Controller is made up by the following classes and interfaces:

- Controller
- IController (interface)
- AbstractController (abstract)
- TickTimer
- TickAttackMain

The Model is made up by the following classes, interfaces, and enums:

- FishingQuest
- FishingRod (enum)
- RangerQuest
- RangerGear (enum)
- Quest (abstract)
- QuestProperties (interface)
- Task (interface)
- TickSearch
- Player
- Item
- ItemFactory
- Store
- StoreFactory
- Tick
- AddGear
- AddRod
- AddRangerItem
- AddFishingItem
- BaseFishingSkill
- BaseRangerSkill
- FishingSkillDecorator (abstract)
- RangerSkillDecorator (abstract)
- PotionType (enum)
- BrewPotion

We decided to enforce the way in which the Controller and the View interact by making use of the IController interface and the AbstractController abstract class provided with midterm 1. By doing this we have a View class that should be able to be easily switched out for an alternative View. Any class implementing the IView interface should be able to function appropriately as a View.

Still focusing on the View, one aspect of the code that may seem to be in partial violation of the MVC framework is that the View works with a reference to the Player class - which is part of the Model. We made this decision because there is a lot of information in the Player that needs to be shown in the View. The alternative to maintaining a reference in the View would have been for the Controller to individually pass along the variables that are tracked in the Player class. However, this would amount to the same thing as passing them all along at once. By using the Player class in the View we also pass along the ability to access all of these variables easily. As such, it made more sense to give the View a reference to the Player class through the Controller. It should be noted that the View does not change the Player class in any way. The reference to the Player class is only used for the sake of accessing values.

The Controller is primarily made up of 2 classes: Controller and TickTimer. We decided to use a java Timer class in order to implement all functionalities that rely on the passage of time. The timer then interacts with the Controller class and sends updates to the View in order to keep the View up to date with the changes that the TickTimer drives.

The remainder of the code that went into our TickAttack game is for dealing with keeping track of game information. These information storing classes do not interact with one another, although some references are maintained between these classes. These references are there either as a result of refactoring, or for structural purposes (or both). For example, the

Store maintains references to Items because the Store is made up of Items. The ItemFactory and StoreFactory classes, on the other hand, came about as a part of the refactoring process to remove the numerous constructors from the Controller and Store classes.

3 Design Patterns

Our implementation incorporates 2 design patterns beyond the MVC architectural design pattern. We used 2 Factory classes in order to simplify the creation of some object types. The ItemFactory class is used to initialise all of the different types of Item, and the StoreFactory is used to initialise each of the 3 Stores that are used in the game. By doing this, constructor code that would otherwise clutter the Controller or some other class is refactored away into the ItemFactory and StoreFactory classes.

The other design pattern that we used in our implementation was the Decorator design pattern. This design pattern was introduced as part of the refactoring process in order to make the addition of new fishing skill and ranger skill boosting items interact with the player in a more intuitive and realistic manner. That is, instead of having the player buy items and have the items affect the player's skill and then disappear, the decorator pattern ensures that the fishing skill (or ranger skill) boosting items are accumulated in a meaningful manner. Furthermore, the decorator keeps track of the 'best' fishing rod and 'best' ranger gear owned by the player. By using the decorator pattern, it made the code very easy to extend. For the last milestone we were able to add in potions that boosted the fishing skill and the ranger skill very easily with no modifications required beyond a line of code to decorate the player with the extra potion where appropriate. This is a demonstration of adherence to the open/closed principle.

4 Other Design Choices

The project as a whole was designed to be easily extendable for any new features that could be imagined for this game. New quests or tasks can be added by creating a new class for the quest/task and connecting them to the controller in the same way that they are for the existing quests and tasks. Again an example of this is given through the addition of the Brew Potion task for Milestone 5.

Within each class the code is designed to follow the Law of Demeter. The interactions between classes is limited to those that would logically be connected and through the controller. For example, the player class does not know anything about any of the tasks or quests other than whether or not they are in the middle of performing one. Instances of applications of the Law of Demeter can be seen in most other classes as well.

The classes and methods within each of the classes are written to follow the Single Responsibility Principle

5 Milestone 5 Additions

6 Testing