Tick Attack Design Report

Finn Lidbetter, Michael Bradet-Legris

November 1, 2016

<u>MVC Design Pattern Adherence:</u>

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

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 in order to remove the numerous constructors from the Controller and Store classes.

Refactoring:

In writing the code there were a number of places where we found the need for significant refactoring using principles learned in class. Some of these steps in the refactoring process will be detailed below.

In our original design we had chosen to have the Tick Search functionality implemented as a quest. However, we found that while the Tick Search shared some similarities with the quests, there were some fundamental differences. We decided that we should make a distinction between ‘Quests’ and ‘Tasks’ such that all Quests are Tasks, but not all Tasks are Quests. This allowed us to implement the Tick Search and Quest functionalities in a manner that took advantage of their similarities but gave the flexibility to handle their differences.

When the item and store functionalities were first implemented we had a lot of code that went into just constructing these objects inside the Controller class. Upon learning about the Factory design pattern in class we realised that we could improve our code by using the Factory pattern to give methods for initialising our stores and items. This removed a lot of messy code from the controller.

Another aspect that evolved in the implementation and refactoring process was the Tick class. We found that we wanted more flexibility in the interaction between a Player and their Ticks than our original design allowed. To remedy this we created a new Tick class that kept track of the information that we wanted to have for a Tick.

One of the areas that needed the largest amount of refactoring using the flocking principles was in the Controller class. Particularly in the process(Object) method. On the first time implementing this code, the way in which to process any Object given by the View was handled in one big method. This has since been refactored into 11 smaller methods. Parts of the processing that were similar for slightly different inputs were brought together using flocking ideas. For example, if you try to buy a RangerGear item or a FishingRod item you need to make sure that you already have the next best rod or gear. Both of these items are defined by Enum types but they are different enums. In order to combine the handling of these purchases into one method, the Enum class was used in the ‘isSkillBoosterAvailable’ method in order to handle the common behaviour of FishingRods and RangerGear. It should be noted that there still seem to be some code smells present in the Controller class - we still have a large switch statement that is dependent on the instruction string given by user interaction in the process(String instruction) method. The purchaseItem method also has an unpleasant looking sequence of if statements that haven’t been refactored out yet.

Similar flocking based refactoring steps were also taken in other places in the code, such as in the TickTimer class, although nowhere needed as much refactoring as was required for the Controller class.

Overall, our code was designed and refactored with the open/closed principle in mind. We believe that it should be fairly easy to switch out our view for an alternative view and that new features such as new quests, new items, or new stores can be added seamlessly.