Drew Chaboyer
William Fiset
COMP 3721

Milestone 5 - Tick Attack Game

How to run game

You can begin by running the game by compiling all the java files and then executing the GameController which contains the main method:

```
~/D/G/A/2/Milestone5 (master) $ ls
AnnouncementView.java IView.java            PotionController.java RecipeController.java testing
Fraction.java         Inventory.java        PotionOne.java        ShopController.java
GameController.java   Item.java             PotionThree.java      View.java
HeaderView.java       ItemFactory.java      PotionTwo.java        ViewDecorator.java
IPotion.java          Player.java           QuestController.java  quests
~/D/G/A/2/Milestone5 (master) $ javac *.java
~/D/G/A/2/Milestone5 (master) $ java GameController
```

Adding new requirements

Completing the first requirement of the assignment was a relatively easy affair. Our pre-existing iteration of the game already had easy item additions. Since players encounter items while completing quests, we added the two new items to a specific quest, this being quest 3. Our .quest files accommodate new items easily, all that is required is to add the item's definition at the bottom of the file, along with specifying which node of the graph should contain this new item. The two new items we added were Zelda's Sword, and a diamond necklace.

The second requirement took much more thought to solve. In our original game, items are designed to be sold by the player. We did not have a feature in place to allow players to consume items, to craft, or even buy them. This required adding several classes, outlined below. We made the decision that it would be onerous for potion ingredients to be part of quests, since that would mean it could take a very long time to accumulate all the necessary ingredients. Our solution was to add a feature for players to buy items at a shop. Once the player has accumulated all the items for a potion they may use their recipe book to craft it. Finally, there is also a menu that allows the player to consume the potions they have crafted to obtain the corresponding boosts.

An issue we encountered was that the items in our game were unique, so our inventory used a set data structure to hold items, but a list makes much more sense because it should be possible to obtain the same item twice. To resolve this problem we redesigned the internal data structure of our Inventory to use the List interface instead of the Set interface.

Design patterns used

With the need for ingredients, and our decision to add purchasing functionality, we saw a strong opportunity to implement the factory pattern. The factory pattern allows easy item instantiation, thus whenever a player buys an ingredient it is very easy to create and place in their inventory.

Our game already had a very strong MVC paradigm presence which we used to our advantage in adding the missing game functionality. Whilst adding the two requirements we added three additional controllers that could 1) buy/sell items in a shop 2) create potions with potion recipes 3) Allow the user to consume potions.

We also have a second explicit design pattern in game: that being the Decorator pattern. This was part of our Milestone 4 work. This added extra flexibility when creating multiple different kinds of views. Our HeaderView and AnnouncementView use the Decorator pattern, as they decorate the base View. This was appropriate because each of them only differs from View by one distinct feature, and there is a lot of repeatable code. The end outcome was that HeaderView and AnnouncementView contained fewer lines of code and were more extensible than if we had employed other methods.

Class Additions

Adding the additional functionality to our code required creating several new classes. The first group of which are the potion classes. As we have learned the value of interfaces this semester, there is an IPotion interface that specifies the mandatory functionality for being a potion. As potions are a type of item, they extend the Item class, but they also have a list of ingredients and an effect they bestow on the player.

Since potions require strange ingredients to create, such as Night Shade, Students Tears, Wolfs Bane, etc… we decided that it should be appropriate to add a shop to our game in order to make it easier to collect these items rather than repeatedly doing quests to grind the required ingredients. For this we added a shop controller as a child

controller of the game controller which is delegated the responsibility of letting the player buy and sell items for their inventory. Below is an example of the player entering the shop, buying Students Tears and returning to the main menu.

```
MENU: shop


WELCOME TO THE SHOP!

Would you like to buy or sell items?
SHOP: buy/sell: buy
SHOP: The shop currently contains the following items:
1) Wolfsbane - 93$ - An ingredient for potion 1
2) Unicorn Hair - 305$ - An ingredient for potion 1
3) Students' Tears - 124$ - An ingredient for potion 1
4) Eel's Eye - 167$ - An ingredient for Potion 2
5) Nightshade - 270$ - An ingredient for Potion 2
6) Dragon Scale - 130$ - An ingredient for Potion 3
7) Hipster Coffee - 131$ - An ingredient for Potion 3
8) Mountie Pride - 123$ - An ingredient for Potion 3
SHOP: You currently have 2500$ you can spend!
SHOP: Please enter the id of the item you wish to buy or 'exit'
Item id: 3
SHOP: You purchased 'Students' Tears' for 124$
SHOP: Your new total is 2376
MENU:
```

To implement the creation of potions, we added a RecipeController class. This class as its name suggests adheres to the standard controller role. It tells the view what to display, in the process listing out the possible potions and their ingredients/effects. It takes user input from the view for which potion to create, and critically only creates the potion if the player has all the ingredients in their inventory. This process consumes the ingredients from the player's primary inventory and adds a potion to the player's hidden inventory. Since potions take a certain amount of time before they become available they are not yet accessible to the player, but they have been created and hence exist in the hidden inventory. Once the player successfully completes a quest we have decided that enough time has passed and the potion should become available to the player, so all the items in the hidden inventory (only potions) get transferred to the primary inventory. Below is an example of creating potion one given that you have all the ingredients.

```
MENU: recipes


WELCOME TO THE RECIPE BOOK!

Recipe Book: There are three potions in this game!
Recipe Book: Potion 1 Description:
Recipe Book: This potion boosts your health, it requires Wolfsbane, Unicorn Hair, and Students' Tears
Recipe Book: Potion 2 Description:
Recipe Book: This potion boosts your money, it requires Potion 1, Eel's Eye, and NightShade.
Recipe Book: Potion 3 Description:
Recipe Book: This potion boosts your health and money, it requires Dragon Scale, Hipster Coffe, and Mountie Pride
Recipe Book: Enter '1' to craft potion one,
Recipe Book: Enter '2' to craft potion two,
Recipe Book: Enter '3' to craft potion three,
Recipe Book: or 'exit' to return to the main menu
:1
Recipe Book: Congrats! You successfully crafted the potion!
Recipe Book: Your potion will become available once you complete your next quest.
MENU:
```

To implement the consumption of potions, we added a PotionController class. As with the other controller classes, it is responsible for decision making. It tells the views what to display, which in this case is the potions the player is capable of consuming. The potion controller then asks the player which potion they wish to consume, applies the potion's effect and removes it from the player's inventory.

```
MENU: potions


WELCOME TO THE POTION USE MENU!

POTION MENU: You currently have the following potions in your inventory:
POTION MENU: Potion 1
POTION MENU: Enter '1' to consume potion one,
POTION MENU: Enter '2' to consume potion two,
POTION MENU: Enter '3' to consume potion three,
POTION MENU: or 'exit' to return to the main menu
1
POTION MENU: This potion makes you feel like a superhuman +250 health!

MENU:
```

Design Decisions

We decided to place quests in their own separate files, which use a standardized structure. This makes it very easy to add new quests. We also opted to use both multiple views and multiple controllers. This multiplicity allows us to have smaller, more manageable classes. Furthermore, it makes the role of each class more specific, a desirable feature in Object-Oriented programming. We also extended this policy to our model classes, those being the player and his/her inventory. By separating those two

ideas we make it easier to extend our code and avoid re-writing pieces that are already in place.

In addition, we adhered to the idea we learnt this semester of using inheritance less and composition more. This is why the main area you see inheritance is with Potions, and with Views. In the view case it is desirable so that we can make use of the Decorator pattern, which at its core uses inheritance. In the potions case, we wanted potions to have all the functionality of items plus more, so we felt justified in using inheritance. In all other cases however, we used composition. For example, our various controller classes frequently have the Player as a class variable. They also have a composition relationship with the views. This allows the controllers to make use of the methods in the player and view classes, without possessing all the complications that come along with inheritance.

Respecting SOLID design

The SOLID design methodology corresponds to a set of design principles which should be followed and incorporated into your software to adhere to good Object Oriented design. We tried really hard to make this happen and along the way we had to change a few things to meet the SOLID criteria. Here are what the five acronyms of SOLID stand for and what we did to meet these principles.

Single responsibility principle:

We setup our classes to have one sole responsibility. This is evident by the names we have given to our classes: Inventory.java, Fraction.java, ShopController.java, QuestController.java GameController.java, etc… Any single class has one and only one responsbility, the ShopController for instance knows nothing about how to consume potions even though it sells potion ingredients, the views only know how to print text and read text, the inventory only manages objects it doesn't regulate what the player can do with them, etc..

Open/Close principle:

We believe our code is very extensible. The decision to extract quests to a separate file means that all one has to do to add more quests to this game is create more of these files. No java code modifications are required. In addition, our decision to have a unique controller for each facet of our game also contributes to its flexibility. For example, when

we added a shop, we simply created a ShopController. Further additions could follow a simple pattern. This compartmentalization allows for easier additions to the game.

Adding smaller features to our game can also be done by extension. The potions showcase this feature, since they extend Item and add additional functionality. By keeping class variables private we also close our classes to modification. One weakness of our code is that some places have branching if statements. This makes them less open to extension than the ideal, however we felt our game had a very adequate level of extensibility.

Liskov substitution principle:

Liskov's substitution principle says that objects in your program should be replaceable by objects of their subtypes without any modifications to your program. If we had not followed Listhov's principle our code would be much less elegant. We do indeed adhere to this principle. Take as an example, the View class and all its derivatives. A HeaderView and an AnnouncementView work in the exact same way as a basic View works except that they display things differently.

Interface segregation principle:

The interfaces we use are small in size, specifying a few methods. This means none of our interfaces attempt to be overly general, thus we obey the interface segregation principle. For example, our IView interface and our ViewDecorator abstract class our two separate pieces. This means you can use one and not be pigeon-holed into also using the other. By making these two distinct entities we have better conceptual separation that allows concrete extensions to only use the pieces relevant to them.

Dependency Inversion Principle:

Since we kept our code very generic, it is flexible. For example, items all function in a similar way, therefore a method expecting an Item as input will execute correctly regardless of the item. This means our methods do not depend on specifics, beyond the inevitable ones. In addition, our code follows this principle because of our well-designed view system. Our code depends much more on the IView interface, than it does its concrete instantiations. The consistency between the various concrete views means that we do not have to rely on one specific one overly much, and we can keep things general by instead relying on the interface. Another place where we have a multiplicity of classes is in relation to items. Here too we did indeed do a good job of making sure

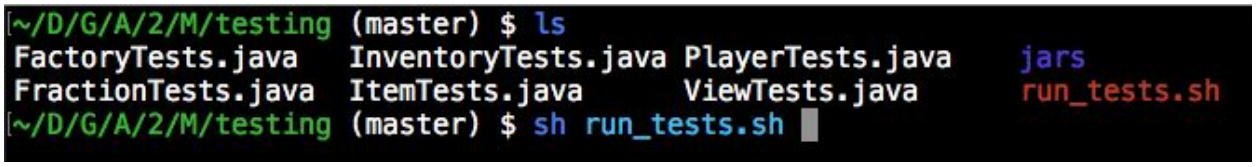
our code was extendable and not overly reliant on concrete classes. Since all items function in the same way, we do not need to depend on any specific item. In addition, the existence of the IPotion interface means that potions have a consistent behavior, therefore we depend much more on the interface than the concrete potions.

Changes in tests

In accomodating to the new requirements we have added new tests in PlayerTests.java and InventoryTests.java. We have also created a new test file FactoryTests.java to test the functionality of the factory class: ItemFactory. The majority of the new tests focus on the interaction between inventory and player because we changed the internal data structure of the Inventory class from a Set to List so it was important to make sure everything still worked smoothly.

How to run tests

The tests for our program can be found in the testing folder. Inside the testing folder you will find various .java test files, a folder of jars containing hamcrest and junit, but more importantly a shell script called "**run_tests.sh**" which can be used to execute the tests for all the various components of the game:



Updated UML diagram

Please view UML.png in our submission to see our UML diagram.