**SE450 Final Fall 2016-2016 Section 701/710**

Name: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Directions: You don’t need to put your name on this exam unless you cannot upload the file to D2L. You may use your computer, blogs, notes, books, internet, etc. Don’t use people for help.

Q1 25

Q2 25

Q3 25

Q4 25

For submission of your exam submit only one zip file.

Inside this zip file submit:

--This document for answers to question #4.

--3 separate zip files for each of your projects

--5 point deduction if you do not meet this requirement.

--You do not need to submit the bin directory for your projects.

--Easiest way to create quickly create these zips is to right click on the project folder and use ‘send to’ compressed file. Open up the zip and delete the bin folder. Now copy these three zip files and the document into your ‘Final’ folder. Send it to zip file. Take that zip file and submit it to D2L.

You’re done!

Name: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Question 1 (25pts)

Download the HomeRunDerby Template and import it.

Using the composite pattern finish the Batter.java class.

From the diagram below:

Your class should use the composite pattern to rank higher and lower batters. If a batter has more homeruns they should go into the ‘higher’ child node, if lower into a lower child node. They do not need to be lists.

You will also need to store outs and homeruns so you can output the winners homerun pitches and the overall winner. Think about how you should do this.

Your class should store the Batters name.

The simulation will randomly assign hit percentages on the pitches and swings. Once you receive the swing object you can use instanceof to test if it’s a homerun or out. Here are the methods you need to complete

void addSwing(Swing swing);

--Takes Swing object, figures out what it is and stores it accordingly

void rank(Batter batter);

--Decides whether the batter has more or less homeruns and ranks it accordingly. Here is where you need to use your composite pattern. If you already have a ‘higher’ batter you can pass it along to the ‘higher’ batter to handle. Remember you have a tree and every node can decide to handle the ranking in its node or pass it on along to a child.

int rankBatter(Batter batter)

--Compares another ‘batter’ with ‘this’ batter object. Just like ‘compareTo’ method. A higher ranking is based off more homeruns. In the case the homerun totals are the same you should call ‘compareTo’ against the batter and ‘this’ name property.

//Overridden method

String toString()

--Prints in one string the entire Batter tree. Highest batters should be first in the string, then current batter, then lower batters. Again think composite (and recursion). Look at the provided output to make sure your output matches.

String getHomeRunDerbyWinner()

--Prints the winner of the derby and the pitches they hit home runs on. Again use the composite pattern. Your root node is not your winner per se. You should see if you have nodes that are ‘higher’.

Here is what your structure should look like.



The output should look like this:

//I will provide this top section of ouput (for all batters, here I’m just showing one batter)

Throwing Pitch #1 : Sinker to batter: Miguel Tejada

It was A HomeRun

Throwing Pitch #2 : Fast Ball to batter: Miguel Tejada

It was An Out

Throwing Pitch #3 : Slider to batter: Miguel Tejada

It was An Out

Throwing Pitch #4 : Curveball to batter: Miguel Tejada

It was A HomeRun

Throwing Pitch #5 : Cutter to batter: Miguel Tejada

It was An Out

Throwing Pitch #6 : Breaking Ball to batter: Miguel Tejada

It was An Out

Throwing Pitch #7 : Knuckle to batter: Miguel Tejada

It was A HomeRun

Throwing Pitch #8 : Change Up to batter: Miguel Tejada

It was An Out

Throwing Pitch #9 : Two-Seamer to batter: Miguel Tejada

It was An Out

Throwing Pitch #10 : Splitter to batter: Miguel Tejada

It was An Out

//You will provide this output

Batter: David Ortiz had 7 homeruns.

Batter: Adrian Gonzalez had 6 homeruns.

Batter: Mark Trumbo had 6 homeruns.

Batter: Joc Pederson had 5 homeruns.

Batter: Josh Hamilton had 5 homeruns.

Batter: Robinson Canó had 5 homeruns.

Batter: Todd Frazier had 5 homeruns.

Batter: Giancarlo Stanton had 4 homeruns.

Batter: Prince Fielder had 4 homeruns.

Batter: Yoenis Céspedes had 4 homeruns.

Batter: Miguel Tejada had 3 homeruns.

Batter: Bobby Abreu had 2 homeruns.

Batter: David Ortiz is the Home Run Derby Winner

. He hit homeruns on pitches:[1][2][3][6][7][8][9]

Name: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Question 2 (25pts)

Download the Automated Beverage Template and import it.

Using the Chain Of Responsibility pattern create three classes in separate java files. IceMachine, SodaMachine,and FlavorMachine.

This simulation will simulate 10 beverage orders from customers. A customer may pick in any order ice, soda, and flavor to create a beverage. The user may only pick ice, or pick any combination up to 3 randomly, but will not pick more than one of each.

Your three classes must extend Automation class. They only need to override processInput. Inside process input you should print out the proper information to match the provided output. You will also need to hook up this method so it continues on in the chain by calling its super class accordingly. It’s one line of code, see the examples provided in class.

The last part is to finish the beverage method in AutomatedBeverage.java. I would recommend you create a reference to a ‘pipeline’ interface of IAutomation as your head of the chain and reverse iterate the ArrayList you receive. As you iterate the arraylist of requests, set the new ‘machine’ objects output to the pipeline then set the pipeline to the machine interface. This way when you are done you’ll be holding the pipeline as the first input machine. Next you need to process the input so it matches the output provided.

\*\*To reverse iterate a BeverageItem array use ListIterator. When you get the listIterator set its parameter to the size of your arraylist. Then call hasPrevious and previous rather than hasNext and next.

//You will provide this output

Assembling Beverage-->[Soda][Flavor]-->Beverage Assembled

Assembling Beverage-->[Soda]-->Beverage Assembled

Assembling Beverage-->[Soda][Ice][Flavor]-->Beverage Assembled

Assembling Beverage-->[Soda][Ice][Flavor]-->Beverage Assembled

Assembling Beverage-->[Ice]-->Beverage Assembled

Assembling Beverage-->[Ice][Flavor]-->Beverage Assembled

Assembling Beverage-->[Ice][Flavor]-->Beverage Assembled

Assembling Beverage-->[Ice]-->Beverage Assembled

Assembling Beverage-->[Flavor]-->Beverage Assembled

Assembling Beverage-->[Ice][Flavor][Soda]-->Beverage Assembled

Name: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Question 3 (25pts)

Download the Immutable Template and import it.

Finish the ImmutableMirroredInteger.java class. Make sure its Immutable and follows all the rules lined out in lecture notes. Follow the comments inside the code. You cannot modify the MutableMirroredInteger.java. I have added asserts to the code to help you verify your class works. See the notes for including this in compilation if it’s not working for you.

Name: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Question 4 (25pts)

Pick 3 design patterns, one from creational, behavioral, and structural respectively. Detail in a paragraph or two each how and when you think you would use the pattern and why. List any strengths and weakness you believe the patterns have. I’m not looking for a text book answer, rather an understanding of things you have learned and how you may apply the pattern to a real problem you face when developing software.

My Creational pattern is the Singleton Design Pattern.

The Singleton Pattern is designed to allow one to be able to limit a class to a single object. This purposeful limitation can an important when such an object is being used over multiple classes and provides a single spot to coordinate the object’s data between such classes. Some pros of using the Singleton pattern is this ease of sharing as creating and managing multiple instances of a class that need to be the same can lead to bugs. Another pro could be that it can be very simple to implement a Singleton pattern as all you need to is make the constructor private and only can access the object through static properties and methods of the Singleton class. A negative of limiting a class to one object can be the fact that one is then limited to that one object even if you might want more. A good example of Singleton use could be if one was trying to simulate crayons in a single shared box of crayons. There is only one of each color crayon so they need to be passed around from child to child instead of each child being able to create a copy of the crayon when used.

My Behavioral pattern is the State Design Pattern

To use the state pattern, one would create objects to represent various circumstances and a context object whose behavior varies as the condition is changed between the various circumstances. One would use such a pattern when the number of conditions and different states increases as the complexity of the code can quickly grow exponentially into a mess and the state pattern helps alleviate this problem. Pros of using the state method are the scalability of adding new states over using a Select-Case approach, the readability of the code stays the same as the number of states increase which results in the reduction of errors. Some disadvantages are that if you have a very low number of states it can be more complex than just using a Select-Case approach. It also takes more memory than a Select-Case approach as each new state uses a new class. A good example of when to use a state pattern is when an object can be in multiple states like a stoplight. A stoplight can be either red, yellow, or green given different conditions and a state pattern could allow one to change the light color as needed.

My Structural pattern is the Composite Design Pattern

To use the state pattern, one would create objects to represent various circumstances and a context object whose behavior varies as the condition is changed between the various circumstances. One would use such a pattern when the number of conditions and different states increases as the complexity of the code can quickly grow exponentially into a mess and the state pattern helps alleviate this problem. Pros of using the state method are the scalability of adding new states over using a Select-Case approach, the readability of the code stays the same as the number of states increase which results in the reduction of errors. Some disadvantages are that if you have a very low number of states it can be more complex than just using a Select-Case approach. It also takes more memory than a Select-Case approach as each new state uses a new class. A good example of when to use a state pattern is when an object can be in multiple states like a stoplight. A stoplight can be either red, yellow, or green given different conditions and a state pattern could allow one to change the light color as needed.

My Structural Pattern is Composite.

The Composite pattern allows a group of objects be treated like a single instance of an object. One would use such a pattern when you want represent part-whole hierarchies or want clients to ignore the difference between compositions of objects and individual objects. Pros of using the Composite pattern are that it makes it easy to add new components to the object and allows clients simpler as they don’t know or care if they’re dealing with a composite component or single object. The con of Composite is that it can make it harder to restrict types of components that can be added to a composite. Good examples of the composite pattern is using a bunch of line objects to create a triangle object made up of lines or using a bunch of node objects to create a tree data-structure comprised of the node objects.