COMS W1007 Assignment 5
Due Dec. 10, 2019

1 Theory Part (50 points)

The following problems are worth the points indicated. They cover some additional patterns, and some topics in advanced Java.

"Paper" programs are those which are composed (handwritten or typed) on the same sheet that the rest of the theory problems are composed on, and are not to be text-edited, compiled, executed, or debugged. For fairness, there will be no extra credit for doing any work for paper programs beyond the design and coding necessary to manually produce the objects or object fragments that are asked for, so text editor or computer output will have no effect on your grade. The same goes for the Theory Part in general: scanned clear handwriting is sufficient, and computer-generated documents will earn no extra credit.

An important note about academic honesty: If you use *any* resource, including `stackoverflow.com`, you must properly attribute the source of your answer with the proper citation. Cutting and pasting without attribution is plagiarism. The CAs will report to the instructor any suspected violations. This even goes for `wikipedia.org`–which for this course often turns out to be inaccurate or overly complicated, anyway–since plagiarizing from Wikipedia is still plagiarism. Much better to read, think, and create your own sentences.

1.1 Proxy pattern (6 points)

Give an example in real life of the proxy pattern, where a system and user first deal with a quick small object to see if the user really wants or is authorized for the real thing. Then show, in a small code fragment each, how each of these Java constructs has a proxy: (1) a class constructor, (2) an ArrayList, (3) a String, and (4) the short circuit if statement.

1.2 Observer pattern (6 points)

Java has a number of different Listeners that interact with Swing components. Start at (the URL below is split; please rejoin before using):

`https://docs.oracle.com/javase/tutorial/uiswing/events/`
`eventsandcomponents.html#many`

where you will find about 20 of them (some on the top of the table, some down the rightmost column). Draw the UML that gives the inheritances for these 20. For example, ActionListener extends EventListener, Is the diagram you drew wide or is it deep (that is, do many listeners extend the same listener, or are there a lot of "levels" of specialization)? What does that mean about the design of Java's listeners?

1.3 Visitor pattern (6 points)

Give an example in real life where the user willingly relinquishes control of user “data” to a “visitor”. Indicate what protections for the “data” that laws or social convention maintain in real life. Then rewrite the four code fragments given on Courseworks so that the small system illustrates this real life case, where the fields and the methods reflect the real life situation. Hint: usually visitors, even if well-intentioned, generate a certain amount of anxiety in a user. Here are two hints, but you cannot use either in you own answer: having surgery, or calling an Uber.

1.4 Shape interface (6 points)

The Shape interface in Java is one of the most important interfaces for implementing GUIs. Look up RectangularShape, which is an abstract class that implements Shape. Name its four known subclasses, including any further subclasses and subsubclasses, etc.,hat extend these four classes all the way down.. One of these further classes, the class Rectangle, is unusual in at least three ways. Find those ways and say what they are. You can get some insight by looking up the class Point, which is similarly unusual.

1.5 Tagging interfaces (6 points)

Some things that look like patterns are not really patterns, but just give instructions to the compiler. For example, “implements Serializable” looks like it might be a pattern, but it is a “tagging interface” that makes no promises about methods at all, and it is unique to Java. Find *three* other tagging interfaces (there at least five in all) in the API (that is, interfaces that have no methods at all, sometimes called “marker interfaces”), and tell what they are used for. Note that the API italicizes interfaces, so this should make the search easier, and to make it easier still, many tagging interfaces end in “-able”.

1.6 Reflection (6 points)

The following code is weird. It will compile, but then run it and see why.

```
Rectangle[] r = new Rectangle[10];
Shape[] s = r;  // since Rectangles are Shapes
s[0] = new Polygon();  // since Polygons are Shapes
```

Why does it compile? What error does it give? Draw the storage diagram that shows what happens at runtime.

1.7 Hashcode (6 points)

Write a hashcode() to handle the class Powerball, which represents a lottery ticket. An instance of this class has a set of six different numbers, but with a particular structure. The first 5 numbers are chosen without replacement from the integers 1 to 69, that is, all 5 numbers must have a different value. The sixth number (the ”powerball”) is chosen from 1 to 26. So, the hashcode() can use a private int[6] array, p, to store these, as long as it satisfies the proper invariant. That invariant is $1 <= p[6] <= 26$ AND for p[i] with $1 <= i <= 5$ then $1 <= p[i] <= 69$ AND for $1 <= i <= 5$, for $1 <= j <= 5$, with $i! = j$, then $p[i]! = p[j]$.

It is not hard to show that there are 69*68*67*66*65*26 possible Powerball tickets, or exactly 35,064,160,560 of them, which is approximately $2^5 * 10^9 = 2^5 * 2^{30} = 2^{35}$. The hashcode() has to map these 6 integers into a single int, even though it is impossible to fit 2^{35} possible tickets into the 2^{32} values an int can provide, even if you find a way to use the negative integers. So, you must find a hashcode() that uses as much of the entire range that is available in an int. That is, you must find a reasonable way to map these 2^{35} values into 2^{31}. Then, show by a concrete example using your hashcode() with real integers, of how it is possible for two Powerball instances to be different, that is, !myPowerball.equals(yourPowerball), but still have identical hashcodes, that is, myPowerball.hashcode() == yourPowerball.

1.8 UML (8 points)

Provide the UML for your Programming part, reflecting the structure of what Steps you actually submitted. Hand-drawn diagrams are OK, and there is no extra credit for machine-generated ones. In each class diagram, you *do not* need to include constants, or getters and setters. But you are required to include all other fields and methods. Make sure your class boxes are properly connected with inheritance ("is"), aggregation ("has"), and association ("uses") links, with any necessary multiplicities. Please, this is the last time you have a chance to use UML to help with your design *before* you start coding!

2 Programming Part (50 points)

This assignment is intended to explore the applet framework. It creates an animation of a kind of RPSKL battle. It uses the appletview of the Java development environment, in order to test the code prior to porting it to some web framework–most of which are unfortunately proprietary and expensive, now that Oracle has abandoned the market.

We require both UML and the usual Javadoc comments within the implementation of your design. You also need to provide test examples, whose output may take the form of a short (20-second or less) video. Please recall that all programs must compile, so keep a working version of each part before your proceed to the next.

Part of this assignment is to give you a small experience of the real world, in which successful code gets maintaining and enhanced. So, this assignment it asks you to use some of what you wrote for Assignment 1. Please follow the Boy Scout rule and clean up what you wrote ten weeks ago!

2.1 Step 1 (10 points): Getting started

Import the Playground.java code that is online in Courseworks. Make sure you properly attribute the source of this code! Also, import Playground.html, which is in Playground.html.txt. Note that you will have to rename that file; the ".txt" is added to keep your browser from trying to run the html.

If you are using Eclipse, find out how Eclipse *simulates* the html file (using the “Run” then “Run Configurations” menu item). Infer from the *.java file what the parameter names and values should be; the values are yours to choose appropriately. Get the Playground applet running in Eclipse.

Alternatively, depending on your operating system, you may be able to run appletviewer directly from the command line, using the html file directly as its parameter.

Verify that the system allows you to adjust the size of the Applet window, and that the start and stop methods actually affect the Timer– although you do not have to prove this by submitting any physical output.

Then, you can now create a real HTML text file, using the template that is in the javadoc of the java file and the example that is in the html. You can do this in Eclipse, if you want, by right-clicking on the package name, then doing “New” then “Other” then “Web” then “HTML”. (You may have to install the Eclipse Web Developer Tools.) Or, you can just use a regular old text editor.

Then move this text html file somewhere on your platform, outside of the Eclipse environment, and copy the java file into the same directory. Then compile the java file, and run “appletviewer”, which should be available as part of your Java environment. You would then say something like this (depending on your operating system):

```
appletviewer Playground.html
```

If you stop at this Step, simply submit the html file you created as proof, and a single screenshot of the Appletviewer window.

Special Note: It may be possible to run this through your own browser. This requires a lot of permission setting in both your system’s Java control panel and the browser’s security settings, and generally is quite complex and dependent on the exact version of Java and the browser. Since systems vary a lot, feel free to exchange notes about how to do this on Piazza. In the past, only Firefox has proven to be easy to get to work properly. Currently (late 2019), it looks like only Internet Explorer 11 and Firefox Extended Support Release (ESR) support Java. See:

`https://www.wikihow.com/Enable-Java-in-Firefox`

Unfortunately, Oracle is gradually withdrawing support for many other Java features on all browsers, due to Java security concerns– including what was supposed to be their ultimate web platform, Web Start. And, every year browser platforms become more proprietary. Rather than require that you learn a new platform for a single assignment at the end of the semester, we will make do with appletviewer. You can experiment on your own with porting your code to a browser if you wish. Surprisingly, the two parts of Java which Oracle promises to continue to support, are the awt and the swing packages–which are two of the *oldest* parts of Java!

2.2 Step 2 (10 points): Two-dimensionality

Using Step 1, make the string movable in both dimensions, x and y, but still at a velocity determined by the HTML. And, the string should be able to start in a location that can be specified in the HTML.

That is, instead of starting at the same place and moving to the left, it can start anywhere within the applet (how do you make sure it is within the applet?) and it can move in nine different ways at a constant speed. Using compass directions, this would be N, NE, E, SE, S, SW, W, NW, or it can stay in the same place. You will have to extend the HTML to allow two parameters for the string that tell it where to start (x and y), and two more parameters that say what the horizontal and vertical velocity direction of the string should be. You can indicate the horizontal as (-1, 0, 1) or as (“W”, “ “, “E”), and similarly the vertical, depending on what you think best. So, the nine directions, starting clockwise at North can be either: (0,1), (1,1), (1,0), (1,-1), (0,-1), (-1,-1), (-1,0), (-1,-1), and (0,0) or “N-”, “NE”, “-E”, “SE”, “S-”, “SW”, “- W”, “NW”, and “– ”.

You will now also have to detect when the string hits any of the *four* borders of the applet, not just the left one. And, at encountering the border, the string should wrap around to the other side of the applet like you saw in Step 1. Be careful of when a string approaches a corner, as it may cross two wraparounds rapidly in succession.

If you do this Step, you do not have to submit separate output from Step 1. But you have to submit your code and either some screenshots or a 20-second video, as you did for Assignment 4.

2.3 Step 3 (10 points): More strings

Enhance Step 2 to allow two or more strings. They all still start where the HTML says, and move at the constant speeds in one of the nine directions according to their HTML, and they “pass through” each other. Pay careful attention to the data structure, and how you structure the HTML. You are allowed to refactor the HTML to make it shorter, for example, if all the velocities are in common, or if all the starting points have the same row or column, or both. You may find the Eclipse html editor helpful.

Most importantly, the strings *must* be visibly different, since many studies have shown that viewers perceive graphics best if colors, textures, and sizes are suggestive of the meaning of the associated text. So, instead of just saying “Rock”, the String should *look* like a Rock (maybe it even says “rOc”), “Paper” should suggest paper in color, etc. Please consider that good GUIs are also good art!

If you do this Step, you do not have to submit separate output from Steps 1 and 2, just the code and a 20-second movie.

2.4 Step 4 (10 points): RPSKL Warfare

Using the strings Rock, Paper, Scissors, Spock, Lizard, detect *collisions* between strings when they are “close enough” to each other when the timer goes off. You will probably want to check in the API for a Rectangle method called “intersects” to get some ideas on how to do this. You will also need to have a data structure that allows a quick check of this “closeness”; you may even want to use some kind of a sort to help you.

Then, apply the rules (and some of the code!) for RPSKL you designed in Assignment 1 to determine which strings survive a collision. So, if “Rock” and “Paper” collide, only “Paper” continues; “Rock” is removed entirely. But if a “Rock” collides with another “Rock”, both should survive. Note that this can get tricky, since more than two strings can collide at once, depending on how you define “close enough”. What happens if two “Rocks” collide with one “Paper”? You might even have to make special rules, such as, if a “Rock”, a “Paper”, and a “Scissors” (or a similar trio) collide all at once, maybe all survive–or maybe none survive. It gets a bit complicated, since you can have simultaneous collisions of very many strings in several different places. And, of course, you must document these decisions and code somewhere. For the sake of debugging, it is probably best to keep this simple.

Note that there are the usual corner cases, too. How many Strings can your window handle? Can they move so quickly that they never intersect at all (due to digitization, can they "hop" over each other)? Can all Strings be stationary, and if not, how do you alert the user? Do you have to worry if two Strings have exactly the same content, location, and movement, so then you see only one? What happens if there hasn't been any collisions for a while–does the display keep running?

Note that there are some simple testing strategies for this Step, since you can easily arrange that some of the strings are stationary. Then they can easily be "hit" by other strings that start off in the same row or column going straight toward them.

As usual, if you do this Step, you do not have to submit separate output from the prior Steps, just the usual code and video.

2.5 Step 5 (10 points): Creative user interface

Do one of the following:

1. Born in Space: The collision of two like objects gives birth to *two* more like objects, each of which goes off in its own direction. This can lead quickly to an overpopulation problem–unless the ecosystem is balanced.

2. Death Star: Use the sample code in MouseTest so that the user can click to "drop" a black hole somewhere in the animation. The black hole never moves. But it destroys anything that comes near to it.

3. A Glitch in the Matrix: Periodically, the animation returns to a prior state, and repeats. This is an example of the Memento Pattern, one of the 23 original design patterns.

Once more: if you do this Step, you do not have to submit separate output from the prior Steps, just code and video.

3 General Notes

For each Step, design and document the system, text edit its components into files using Eclipse, compile them, debug them, and execute them on your own test data. When you are ready, submit the text of its code, a copy of your testing, and whatever additional documentation is required. Make sure that the source code is clear and its user interface is comprehensive and informative.

Write your classes clearly, with a large block comment at the beginning describing what the class does, how it does it (i.e., in some form, talk about its design assumptions), and justify how it will be tested. Clear programming style will account for a substantial portion of your grade for both the Theory Part and the Programming Part of this assignment. For one reasonable set of suggested practices and stylistic guidelines (but one that does not refer to the Eclipse defaults), see the website of the textbook used in COMS W1004, at: `http://www.horstmann.com/bigj/style.html`

4 Checklist

Here is an approximate checklist for submission. Please note that this is designed to be a *general* guide, and you are responsible for *all* things asked for in the text of the assignment above, whether or not they appear below. *Please check with Piazza* to see if there are any additional requirements or suggestions from the TAs.

4.1 For the Theory Part

Answers, either handwritten then scanned, or in machine-readable form. No extra points for using a text editor or other formatting tools; hand-drawn diagrams are fine. Submitted electronically in Courseworks by the time specified (i.e., by the beginning of class, exactly). The theory file should be separate from the files for the programming. Name and UNI must be prominently included.

4.2 For the Programming Part

Answers submitted electronically in Courseworks by the time specified (i.e., by the beginning of the review session, exactly). All code (*.java files) documented according to Javadoc conventions and all trial runs. These files should be separate from the files for the theory. Name and UNI must be prominently included in each class (use "@author") and on each output. Rather than providing a separate ReadMe file, one of your classes should have an obvious name that will attract the attention of the TAs (like "Playground.java", for example, since Applets don't have runners), and *inside* it there should be an overview of the components of your system. The classes, in whatever state of completion they are, *must compile*. They must also produce the output that you submit; the TAs will penalize and report any mismatch between what the code actually does and what you say it produced. If you include a video, it must be no longer than 20 seconds, even if it has been deposited somewhere other than your submission.

4.3 For both Parts

Please put all your submission files (theory, *.java, example runs) in one directory (named, for example,"HW5"), and then compress the contents of that directory into one file. Submit that compressed file to Courseworks. The name of that compressed file should be "myUNI_HW5", and it should have the appropriate extension like ".zip", except that "myUNI" should of course be replaced with your real Columbia UNI. *Please* use this convention as it makes the TAs' job much easier:

You may submit electronically as often as you wish, but it will be your *last* electronic submission before the deadline which is the official one that will be graded and (if need be) executed.