COMS W1007 Assignment 4
Due Nov. 27, 2018

1 Theory Part (48 points)

1.1 Pattern: Builder (6 points)

Explain, and give an example, why the Law of Demeter is sometimes called the "use only one dot" rule. Then explain, and give an example, of how the Builder pattern from Assignment 3 uses many dots, but still does not violate the Law of Demeter, and is therefore usually not called a "train wreck".

1.2 Enums (6 points)

Enums come with some unusual methods. Look up, explain, and give an example of what compareTo() does. Explain and give an example of these two methods that are nearly the same, and explain how do they differ: name() and toString(). Lastly, there is a "secret" method, values(), that is not part of the API definition: is this a bug or a feature? How does this method relate to the "varargs" construct used in passing parameters?

1.3 Pattern: Iterator and Composite (12 points)

Java supports multi-dimensional arrays. Suppose you have a two-dimensional int array that is "jagged" and not "square". Write a method that sums all the values in it. Now, imagine it is three-dimensional and jagged, not "cubic". Your previous method will get messy, so rewrite it in the style of the Composite pattern so that it can even handle higher dimensions. (Consider the array to be boxes of boxes of ... integers.) Note that you do not need any interface here, but you will need to look up "isArray()" in order to handle the "leaf" nodes of this pattern.

1.4 Patterns: Composite and Null Object (7 points)

In the Courseworks example about Stuff, what would be the "Null Object"? In terms of the real world user problem (that is, with boxes and bubblewraps), where should you be able to use this null object? Given an example. Then, using the lecture example of <<ListItem>>, do the same: what would be a Null Object, and where in the real world have you seen it (Hint: look up "S&H")? Lastly, explain why the <<ListItem>> doesn't need a new class called "TaxedItem".

1.5 GUI (7 points)

The Java API give seven constructors for the class Rectangle. Notice that they combine ints, Points, Dimensions, and even Rectangle. Some other possible constructors are missing. Using the seven existing constructors as a basis, list *all* the possible missing constructors which would make sense

using ints, Points, and Dimension. Note that Dimension is very strange: How is it different from Point? Is there any clear reason why it takes ints in its contructor, but returns double from its getters?

1.6 UML (10 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.

2 Programming Part (52 points)

This assignment asks you to apply the principles of good class construction to a simple interactive console GUI program using the AWT and swing packages of the Java API, and various patterns and interfaces, particularly Composite.. We do not require CRC, but we do require UML as one of the questions of the Theory Part, 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 screenshots or a short (20-second or less) video. (You can post the video to some depository and then give the URL for it.)

Please recall that all programs must compile, so keep a working version of each part before your proceed to the next.

2.1 Step 1: Getting started (20 points)

Create a simple GUI display that shows a simple unicycle moving from left to right at a constant speed in a JFrame. The unicycle is stylized as two nested circles (the wheel), a line (the frame) and a rectangle (the seat), crudely like "O-#" except upright, and with the line going to the center of the circles. Use Step 2 in the following tutorial, except that the seat is rectangular:

`https://www.drawingtutorials101.com/how-to-draw-a-unicycle`

Download some suggestions for code segments from Courseworks. Make sure you properly attribute the source of this code! As those are only just rough suggestions, make sure you modify the Javadoc and comments appropriately, according to the Boy Scout Rule,

Make sure that the unicycle is drawn using *relative* coordinates, rather than absolute ones, since further Steps will require the reuse of the instances of unicycles with different sizes and locations. This means you need to minimize the use of magic numbers. Instead, you should express all sizes and locations in terms of a UNIT (which could be, for example, the length of the frame or the height of the seat), and any mutual relationships (wheel versus frame, frame versus seat, relative shape of the seat) in terms of CONSTANTS. See the code suggestions on how to organize these x and y values.

To display your output, capture some screen shots, which you include in your zip file, or record with your cellphone a video of no longer than 20 seconds, which can you include in your

submission, or deposit in a publicly available forum with the appropriate URL. Please note: If you use a video, if it is longer than 20 seconds, it will be *ignored*, and you will not get any credit for it. You can decently get by with just 10 seconds anyway

2.2 Step 2: Refinement (16 points)

Using Step 1, make a kind of "self-driving unicycle race", like the following, except that your unicycles are *self-driving*:

`https://www.youtube.com/watch?v=8lOJQ_nUKro`

This would consists of a small number (from 3 to 7) of unicycles of random sizes (limited to 0.5 to 2.0 times the usual UNIT), in random places relative to the horizontal axis and to each other (again reasonably limited, in terms of UNITs). They can be drawn "on top of each other"; that is, you can fill() in the "seat" so that they don't look transparent, but you should be able to "see through" the open space where the wheel is. This shading helps establish the illusion of who is in front, since the seat will "block out" anything behind it..

Then, add a JSlider and its ChangeListener to control this group of unicycles so that they all move together as a group in a horizontal direction at the same constant speed (i.e., no unicycle is overtaking any other). You might consider using GeneralPath again, to make the construction of this group easier. The slider should be displayed horizontally, and it controls the speed, whether to the left, to the right, or even stationary (but be careful here, as you might encounter an ArithmeticException if you attempt to divide by zero).

As usual, if you do this Step, you do not have to submit separate output from Step 1. Again, any videos must obey the restriction of 20 seconds in length.

2.3 Step 3: Creativity (16 points)

Using Step 2, pick *one* of the following enhancements, which should be straightforward once you have the infrastructure above. If you do this Step, you do not have to submit separate output from Step 1 or Step 2. However, you do also have to show the use of an additional *interface* that you write yourself. Usually this will be some kind of Composite pattern interface. This must be clearly documented so that the TAs understand how you used design methods to modify the original specifications given in Steps 1 and 2 by introducing this interface. You can get some inspiration from the example on Courseworks using the interface Stuff.

1. Use the Composite Pattern to group some unicycles into packs that move according to the JSlider, but each group has its own speed multiplier. For example, one group moves exactly like Step 2, but a second group moves 20% faster. If you do this properly, you can have groups of groups, so that you can have a large crowd overtaking a smaller crowd, but within each crowd there is some further differences in speed.

2. Two (or more) separate packs each have their own horizontal location and slider. But the sliders are sensitive to size, that is, the slider that controls a bigger unicycle has more effect, since a "bigger" unicycle is actually closer in 3D to the viewer. There is a simple mathematical rule for doing so; look up "one point perspective".

3. Add some up-down "jitter" to the motion of each individual unicycle, so that horizontal progress is not longer straight, but appears "bumpy". The interface can now define a group of unicycles seeming to go up and down some bumpy roads.

4. Add pedals to the wheel. These would just be one line through the wheel center, sort of what you see when you write "0" in math style with a slash through it. (These parts are technically called cranks.) Have these pedals rotate as the unicycle moves. Try to match crank rotational speed to wheel forward speed; the math is pretty simple to do, without having to determine it experimentally. If you want to add actual pedals, they can be simple filled rectangles at the ends of the cranks, which are always aligned with the x and y axes, regardless of the crank orientation.

5. This one is quite tricky, but aesthetically pleasing. Modify the apparent geometry of the JFrame to make the GUI window appear infinite, by making the unicycles get progressively smaller as they approaches the border, so that they never really reach it. This is not as hard as it sounds. This is called a "hyperbolic space", and is sometimes used to display the contents of large databases. See:

   `https://www.youtube.com/watch?v=8bhq08BQLDs`

   or look up Escher's woodcuts in his "Circle Limit" series to get a general idea, at:

   `https://www.google.com/search?tbm=isch&q=escher+circle+limit`,

   or you might want to look up "Poincare disk model wiki". The interface here would generalize the concepts of drawing something at (x, y), depending on whether the geometry is the usual Euclidean one, or a hyperbolic one in the y dimension (it will look like a cylinder on its side), or a hyperbolic one in both x and y dimensions (it will look like a sphere).

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. For this assignment, which is fundamentally about GUIs, AWT, and swing, we do not require CRC, but we do require UML.

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 class, 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 "HW4Runner", for example), 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.

Please note: 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,"HW4"), 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_HW4", 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.