COMS W1007x: Object-Oriented Programming & Design (Java)
Fall, 2011
Assignment 4: Due Tuesday, November 29, 2011, 1:10 pm, in class

Part 1: Theory (45 points + 6 points left over from Ass. 3)

The following problems are worth the points indicated. Except for the question left over from Chapter 5, they are generally based on the book's exercises at the end of Chapters 6 and 10, under the section heading "Exercises". They cover issues in inheritance, abstract classes, and patterns.

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

Assignment 3: 5) (6 points) Generalize 5.14 but do not write a paper program. That is, the question becomes: what problems can occur when there are more than one decorator that all use methods of the same base component? Hint: why do bold, italics, and underlining work as decorators of a font, regardless of their order of application? Further hint: think about sequence diagrams. How can these problems be fixed in general?

1) (10 points) Exercise 6.2. Write a paper program defining the two classes, but do not add the book's suggested method; add a method getAverageMonthlySalary() instead, based on reasonable assumptions of the days in an average month.

2) (5 points) Exercise 6.8, mostly: give two examples of final methods, two examples of final classes, and two examples of final public fields. Unfortunately, there is no consistent use of these in Java, so you will need to hunt around in the API. Lastly, is it possible to have a final interface? A final abstract class? Explain.

3) (5 points) Exercise 6.10. Use the Java API for help.

4) (5 points) Exercise 6.18. Justify your answer in two ways. First, by showing how each of the four properties of the pattern you chose (name, context, problem, solution) is implemented by this class. Second, by drawing a UML diagram like those in Chapter 5, except that you label the classes with specific class names and methods.

5) (5 points) Like exercise 6.20, except do it for SpinnerNumberModel (a rather interesting if a bit bizarre class) and addChangeListener. Look this up, again, in the API.

6) (5 points) Exercise 10.2. Show the properties that Adapter pattern requires, and indicate which ones are missing. Then, explain what pattern(s) these "adapters" are examples of instead.

7) (5 points) Exercise 10.6. This is necessarily an essay question, but your answer should focus on the concept of "state".

8) (5 points) Here is a real life situation. Say what pattern it is, explain how it meets each of the name / context / problem / solution slots, and draw the UML diagram for the concrete situation (not the universal abstract one). The situation: People need transportation. Some prefer subway, some prefer buses, some prefer cabs. Each person knows the process for asking for the kind of transportation service they prefer. However, once they start that process, they have no further say in how that process happens, they literally have to go along for the ride. But for their part, the subway, bus, and taxi companies are free to change at any given time exactly what steps will comprise the delivery of their services, even without informing the people who will ask for them.

Part 2: Programming (55 points)

This assignment is intended to explore inheritance, abstract classes, and patterns.

Please recall that all programs must compile, so keep a working version of each part before your proceed to the next. Please also note that soft-copy Javadoc output is required, including class comments, constructor comments (with @param) and method comments (with @param and @return, as appropriate). You also need to provide some test examples: make sure you turn in some screen shots to show the functionality of your system.

Step 1 (20 points): Getting started. Implement the code in section 6.3, pages 232 to 234. The source code can be downloaded from the book's on-line companion. Make sure you properly attribute the source of this code!

However, you will find that by itself it won't compile, as it needs the interface on page 228, the abstract class on page 229, and the concrete classes on page 229, most of which happens to be implemented using the Template pattern on pages 238 and 239. But, you will probably need to modify some of the code from pages 168 to 169 to get CarShape (which you used in Assignment 3). Some of this code is available in a related form as the answer to Exercise 6.17 in the on-line companion, although the code that you will find there answers a somewhat different question and is not quite what you need.

Then, just as you did in Assignment 3, you should rearrange the code and remove any magic numbers so that whatever you import is easier to understand and is more self-documenting. If you wish, you can redesign the look and feel for both car and the house to personalize them or to make them look more attractive, but you won't get extra credit for doing so. Test the code and submit some evidence of your testing.

Step 2 (17 points): Using Step 1, do Exercise 10.8. That is, using the Action pattern, and the code on pages 405 and 406, change the buttons you used in Step 1 so they can be enabled and disabled. You must decide with each action when they should be enabled or disabled: you should have both a lower limit and an upper limit on the number of houses and cars--either separately for each, or their number in total taken together--that can be drawn or removed. Note that these buttons can therefore interact with each other: for example, a disabled button can be re-enabled by the action of a different button. These decisions should be documented in your design and code.

If you do this Step, you do not have to submit separate output from Step 1.

Step 3 (18 points): Creativity Step. Add an enhancement similar to those if Exercise 6.14, except that your enhancement can't be a truck or a stop sign. You only need to do one new extension of

SelectableShape, and it can be simple and small--but it must have at least two primitive shapes that make it up (that is, it must use GeneralPath). It should also have its own Action pattern "Add Me" button, and be controllable by the "Remove" button. And, it should establish and/or affect the minimum and maximum limits on how many of these shapes, as well as the existing shapes, the buttons can control. For example, it could be that this new shape is a kind of car, so the total number of cars permitted obeys one rule but houses are independent. Or maybe it is so complicated that it counts as two shapes. Or maybe it can't be drawn unless some other condition is met, like, the number of houses exceeds the number of cars. Whatever you decide, you must document it clearly.

But in addition: this new shape must have an extra property different from the House and Car shapes. It can be drawn anywhere at all, even in a fully random location, except that it *should not* overlap any existing shape. Note that sometimes this means you cannot draw the shape at all(!), and your testing should show this.

There are many ways to do this, and most of them require looking at all the existing shapes and using the GeneralPath method "contains". And, some of these algorithms are not going to give exact results all the time, depending on how the algorithm represents the new figure and how it compares this representation to the existing shapes using "contains". Not that "contains" is required in the Shape interface in several forms, and that therefore it has been actually implemented for each specific shape(s) in the API.

Make sure you clearly specify in a comment what ever algorithm you have chosen, and explain whether it is exact (that is, whether it is guaranteed to work always) or only approximate (that is, it depends on a heuristic or on an approximation, and may occasionally fail as the frame fills up). If it is approximate, clearly explain what can cause it to fail. Note that you do not have to use an exact algorithm as long as you clearly indicate the strengths and weaknesses of an approximate one.

If you do this Step, you do not have to submit separate output from Step 1 or Step 2.

General Notes:

For each step, design the system using whatever CRC, UML, or other technology you find helpful; we do not require you to submit the results of these tools, but you should find that them make the Javadoc easier. Then, write the documentation and the system, and text edit its classes into files. Compile them, execute them on your own test data, and debug them if necessary. When you are ready, submit a hard copy of the text of the code and your testing (use appletviewer or some other screen capture tool), and whatever additional documentation is required; submit an electronic copy of the your class and javadoc files. 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 CRC and UML), and how it will be tested. Describe the test data in the comment, also. Document each constructor and method, using javadoc tags and other commentary as necessary. Clear programming style will account for a substantial portion of your grade for the programming part of this assignment.

Checklist for submission:

For theory: Hard copy, handed in to Prof or TA by the beginning of class. No extra points for using a text editor. Neatly stapled, separately from programming hard copy! Name and UNI attached.

For programming: Hard copy, handed in to Prof or TA by the beginning of class. All code, all testing runs (cut and pasted from console, and/or captured screenshots). Neatly stapled, separately from theory hard copy! Name and UNI attached.

Also for programming: Soft copy, submitted to the Course Files Shared Folder on courseworks by the beginning of class, in a tarball created by CUIT Unix tar command ("tar -cvzf uni_HW4.tar.gz whateverMyDirectoryIs"). Please use "uni_HW4" as the name of your Courseworks submission. Include softcopy of javadoc html. Name and UNI should be included as comments in every class. The code must compile. Last submission before deadline is the official one that will be executed if needed.