COMS W1007x: Object-Oriented Programming & Design
Fall, 2011
Assignment 1: Due Tuesday, September 27, 2011, 1:10pm

Part 1: Theory (50 points)

NOTE: At the time of this writing, the class has been assigned only 3 TAs, so it has been made deliberately shorter than usual.

The following problems are worth the points indicated. They are generally based on the book's exercises at the ends of Chapters 1 and 2, under the section heading "Exercises". They cover Java fundamentals and internals, plus the basics of object-oriented design.

The instructor is mindful of the necessity for learning Unix, Emacs, and Java infrastructures, and has taken this cultural transition into account. This assignment therefore gives more credit than will be usual for the programming section, and it has more interconnectedness between the Theory Part and the Programming Part.

"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 Part in general: clear handwriting is sufficient, and computer-generated documents will earn no extra credit.

1) (6 points) Aspect oriented design: Start at the excellent tutorial found at http://www.developer.com/design/article.php/3308941 online. Then give three examples of "aspects" of design that object-oriented design does not address. Then, say what "advice" is. Then, say what a "weaver" does. Then, give a convincing reason, from a design perspective, why aspects have not become part of Java.

2) (6 points) API familiarity: Find the official Java API (official name: "Java(tm)Platform, Standard Edition 6 API Specification"). Note that Java is owned by the Oracle Corporation, and note that the official API is still Edition 6. Bookmark it; you will be using it a lot in the course. Copy as part of your answer its URL. Then, give the names (just the names) of the fields, constructors, and methods, complete with their parameters and return values, for the class Locale. You do not need to give any text descriptions, nor any of those fields or methods that have been inherited from any superclasses. This is meant to be an easy question, to get you familiar with the API; just copy down those parts requested. But now, answer this: What does this particular class say about the design of Java: in what ways is it good, or bad, or both?

3) (6 points) Basic data types: Suppose you are trapped in a time warp of 60 years ago and find yourself working for the Univac Corporation. You have decided to organize your computers around a 36 bit "word". (This really was the case.) Estimate the range of the integers you can represent in a halfword, assuming ones-complement. Since they really did use ones-complement, how did their integers differ from today's standard?

4) (16 points) Simple console system: "Write" a simple Wind Chill Index calculator that is executed by providing on the command line its run-time parameters (in the English system of weights and measures), like:

```
$java WindChillIndex temperature windSpeed
```

that is, a simple use would be:

```
$java WindChillIndex -5 20
```

This small system requires the use of String[] args, and conversion routines. And, in keeping with OOD, you should have a "worker" class and a "tester" class. You will probably have to use the web to find out the formula; try the National Weather Service.

Make sure you handle any constants properly, in keeping with good design practice. Please be aware that the NWS says that wind chill is not defined for some temperatures and some wind speeds. So you have to find out what these are, and then decide how, where, and why your design accommodates these important restrictions.

Write a "paper" program: do not text edit, compile, execute, or debug, and do it on the same answer sheets as the rest of this Theory Part. However, unlike most other theory assignments, you will later be asked to *execute* an extended version of it as part of Part 2, as an exercise in learning CUIT Unix and Java--and also of using object-oriented design and reuse.

5) (8 points) CRC for the programming part: Sketch the CRC cards that you developed for Step 2, 3, or 4 of the Programming assignment, depending on how far you got. These can be free-hand, on standard paper; you don't have to use actual 3x5 cards. Please do these *before* you start coding! You only have to turn one set, in the final form that documents whatever you developed and submitted in the Programming part.

6) (8 points) UML for the programming part: Show the UML diagrams that you developed for Step 2, 3, or 4 of the Programming assignment, depending on how far you got. These can be free-hand; no extra credit for using Violet. Please do these *before* you start coding! You only have to turn one set, in the final form that documents whatever you developed and submitted in the Programming part.

Part 2: Programming (50 points)

This assignment is intended to walk you through the mechanics of CUIT, Unix, Emacs; to remind you of the class/object/method approach of Java; and to exercise CRC, UML, and other design tools to develop a simple system and then augment its functionality.

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

Step 1 (10 points): Getting started. Unlike future homeworks, this particular Step is not related to the following Steps; basically, this is an immigration exercise. Simply implement the Wind Chill Index program of the Theory Part, except that it takes *console* input from a user, not command line input, by asking for the two components of the calculation and outputting the answer. You should have it repeat these requests to the user, until some certain input sentinel will signal that the program should exit.

This is primarily to get you started with something that you know works, to give you the infrastructure for writing console input, and to get you acquainted with how to do it on the CUIT systems. Again, you should have a "worker" class and a "tester" class. Turn in a listing of your code, and some trial runs.

Design note: If you did Question 4 in the Theory Part properly, all you should need to change is the "tester" class; the "worker" class should be absolutely untouched. This is one of the reasons for having OO Design!

Step 2 (18 points): An exercise in simple design. Build a (much) scaled-down version of a hamburger stand. You get separate and additional credit for this Step, and if you do it, you should submit it in addition to Step 1.

Design a simple object-oriented system that uses console input and output to take an "order" from a user, then builds a "burger" and computes and reports the "cost". The quotation marks are necessary because none of these objects are real; here they are just Strings and primitives.

The design will develop in a sequence of steps, so you might want to read through all the steps first, so that future enhancements are done smoothly and transparently. (But most designers of working systems don't have this luxury, of course!)

For the first design, you need a BurgerDude class (Horstmann would call the class BurgerTester) that sets up the other classes, which are a CounterPerson class and a Cook class.

At this Step, the CounterPerson simply asks users a fixed series of binary questions to determine what should be included in the assembled burger. This information eventually (directly? indirectly?) ends up at the Cook, which constructs the burger (which is really just a String) and computes the cost (which is really just a primitive). The CounterPerson then outputs this information. Of course, users can make mistakes talking with the CounterPerson, so there has to be some error checking and error handling--which should be documented.

For a ridiculously short example that can be cleaned up *considerably*:

```
BurgerDude> Order your burger by indicating yes by "1" and no by "0".
BurgerDude> Do you want an extra meat patty?
0
BurgerDude> Do you want cheese?
1
BurgerDude> Do you want ketchup?
3.141592
BurgerDude> I take that as a no.
BurgerDude> Here is your burger.  It has cheese and it costs $2.50.
```

[Note that this system is a bit too much in the original spirit of Java, in that it assumes a single CounterPerson and a single Cook. Real BurgerDude hamburger stands would have several of each working in parallel.]

Make sure you document within each class exactly what processing happens, since in later Steps it will become more complicated. Take care with designing any constants that you use, since in the next Steps you may eventually incorporate them into a class. But for this Step, they can be Strings and ints, respectively. (You do realize that floats or doubles are a big mistake for items that can be counted, right?

You should count in ints of cents, not doubles of dollars!)

You also have some "user interface" issues in the CounterPerson you should resolve: how to present the text describing how to use the system; how to organize and "display" (just on the screen as text, without using fancy graphics) any input, results, or error messages; how to allow users to indicate they are done even before the fixed series of questions is complete; whether to be "impatient" with annoying users (for example, users who keep inputting negative amounts); what to do on user input errors like alphabetic characters; etc. Wouldn't it be a good idea for any user to be able to order more than just one burger, too?

Please note that it will make the design and implementation of this Step much more pleasant if you do the CRC cards and the UML diagram *first*. You will have to turn them in as exercises in the Theory Part anyway, so please put the time and intellectual effort up front! At this Step, you should have just three classes.

In addition to code listing and trial runs, you should also run javadoc on your package of classes and turn in a sample of the documentation generated.

Step 3 (12 points): An exercise in object reuse. Make your system more realistic by having two more classes, one called BurgerPart, and one called Menu. A BurgerPart records the name of an item and its cost, for example, "extra patty", "100". A Menu records which BurgerParts are offered today, and how many of each is legal for a user to order. For example, a user can order no more that two extra patties, or no more than 17 hits of ketchup, or maybe even no pickles at all if somehow the stock has run out. Why do you want to do this? Because things may change, some BurgerDudes may offer different condiments, prices can go up or down, etc., and so the design should acknowledge the possibilities of future variations all in one place (one class).

It is your job as a designer to decide how these classes are created and integrated into your existing system. Clearly, Menu should work somehow with CounterPerson, and BurgerPart with Cook. You might even consider if it makes sense to have one more class called Order.

Note that this Step should help illustrate the ways in which a object-oriented design--including the CRC and UML tools--makes increasing the system functionality easy (or hard!). Please go back to your CRC cards and the UML diagrams *first*, and think out your design using them. You will be getting credit for them in the Theory Part anyway.

If you do this part, you still have to submit your output from Step 1, but not the output from Step 2. The usual output is required: code listing, trial runs, javadoc output.

Step 4 (10 points): Creativity Step. Further stretch the boundaries of your object-oriented design. If you do this part, you still have to submit your output from Step 1, but not from Steps 2 or 3.

Improve your system in one of the following ways. Note: you will only get credit for doing *one*, so don't try to do more. It is critically important, however, that you clearly specify (Specify it where? Internally, of course!) what you have decided to do, and why.

Note that this Step should further illustrate the ways in which object-oriented design tools actually work.

Please go back to your CRC cards and the UML diagrams *first*, and work out your enhancements using them before going back to your code. Usual output is required: code listing, trial runs, javadoc output.

Pick your single choice of one of:

a) Get fancy with "specials". You may want to encourage certain kinds of consumption by having lower prices on certain combinations of BurgerParts. For example, the second additional patty might be at half price. Or, if you order bacon, the cheese comes free. Where should this information go? A good design will encapsulate these relationships so that existing classes do not have to be redesigned much. How will you let the user know what the specials are? Do you need a Special class?

b) Have concrete limits. A user should be able to order multiple burgers, but sometimes the hamburger stand can run out of something, like avocado. Augment your design so that at the beginning of the day the hamburger stand has a certain inventory of each BurgerPart, which is appropriately decreased with each order. If your design is a good one so far, this additional functionality can be handled by additional methods rather than by additional classes. Unless, of course, you want to design a Limit class.

c) Allow the manager to change the menu on-the-fly. Your system can have a secret security code that alerts the counter person that the user is now really the manager, and he has just bought some bean sprouts and jalapenos that should be included in all future menus. How should your design handle this gracefully? Does it matter where in the Menu these new BurgerParts go? Should you create a Manager class?

The usual output is required: code listing, trial runs, javadoc output.

General Notes:

For each Step, design and document the system, text edit its components into files, 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 CRC), 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, see the website of your textbook's author, Cay Horstmann, at:

`http://www.horstmann.com/bigj/style.html`

Checklist:

Here is a 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.

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! 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), all javadoc if javadoc is required. Neatly stapled, separately from the theory! 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_HW1.tar.gz whateverMyDirectoryIs"). Includes softcopy of javadoc html. Name and UNI included as comments in every class. The code must compile. Last electronic submission before deadline is the official one that will be executed if needed.