Computing & Software Systems 162:
Programming Methodology
Spring 2009

Basic Course Information

In this course, you will transition from a focus on basic programming skills to applying those skills to solve problems. You will do this by learning to think of *software development* as a rigorous process, in which the actual programming is one of the smallest parts. You will be introduced to higher-level problem solving approaches, such as recursion and generic programming, and larger-scale organization and algorithms, like object orientation, lists, stacks, queues, searching, and sorting. You will gain familiarity with software development techniques, such as the importance of thinking about specifications, design, and testing before coding and the utility of incremental development in an exploratory environment. You will also develop an understanding of the mathematical nature of software development by examining the relationship algorithms, programs, and the underlying theory, including logic, sets, functions, number bases.

Lectures Mondays and Wednesday, 1:15–3:20PM, room UW1-202.

Laboratories Mondays 3:30–5:35PM, UW1-120.

Instructor Michael Stiber stiber@u.washington.edu, room UW1-360D, phone (425) 352-5280, office hours Wednesdays 11AM–noon or by appointment.

Course Web http://courses.washington.edu/css162/stiber/.

Required Textbooks Walter Savitch, *Absolute Java*, 3rd edition, Addison-Wesley, 2007.

Kenneth H. Rosen, *Discrete Mathematics and Its Applications*, 6th edition, McGraw Hill, 2007.

M. David Ermann and Michele S. Shauf, *Computers, Ethics, and Society*, 3rd edition, Oxford University Press, 2003.

Suggested References Arnold, Ken, James Gosling, and David Holmes, *The Java Programming Language*, Fourth Edition, Prentice Hall PTR, Boston, 2005.

Sun Microsystems, The Java Tutorials, http://java.sun.com/docs/books/tutorial/

Class meetings and you I consider you to be adults and I will treat you as such. I therefore do not take attendance and leave it up to you to come to class or not and to assume responsibility for the consequences of your decision. However, I *strongly* encourage you to come to class and, in fact, a portion of your grade will depend on your participation in in-class activities and labs. You will be held responsible for *all* material covered in class, regardless of its presence (or lack thereof) in the textbooks. The same goes for announcements and clarifications of homeworks, etc.

While in class, I ask that you not engage in behavior that may be disruptive or distracting to your colleagues. In particular, if you plan to use a computer during class, please sit at the rear of the class. In my opinion, computers are not the best way to take notes.

Grading Your grade will be composed of your performance on tests and homeworks, plus your classroom contributions as measured by lab reports and other in-class activities.

Both the midterm and the final are equally weighted. The weight of each homework will be governed by the number of points assigned to it; generally, each program will be worth 100 points and each written assignment will be worth around 25–30 points. I reserve the right to assign a different number of points (and thus a different weight) to specific homeworks. I will compute your homework average

by summing the number of homework points you earned and dividing by the total possible. Laboratory reports (typically, done in groups) will be graded pass/fail.

Your course average will be computed as: 25% homework + 30% midterm + 30% final + 15% labs.

If you do significantly better on the final than the midterm, I will use the alternative formula: 25% homework + 25% midterm + 35% final + 15% labs. So it is *possible* to get off to a slow start and recover, but I don't recommend that you plan this as your strategy for the quarter.

I don't grade on a curve. I compute everyone's quarter average based on the formula above. I then use my judgment to determine what averages correspond to an 'A', 'B', etc. for the quarter. Some quarters' assignments, etc. turn out harder, and so the averages are lower. Other quarters, assignments are easier and averages are higher. I use my judgment to adjust for that at the end. Decimal grades are then computed using the equivalences in the *UW Catalog*, linearly interpolating between letter-grade boundaries. Furthermore, I am well aware of the significance of assigning a grade below 2.0, in terms of impact on your career here at UWB. I can assure you that I examine *in detail* the performance in this course of each student before assigning a grade below 2.0.

What is the difference between this and grading on a curve? With the latter, the goal is to have X% 'A's, Y% 'B's, etc. My way, I would be happy to give out all 'A's (if they were earned). A shorthand summary of the qualitative meaning of letter grades is:

A Complete or near-complete mastery of all course subject matter. Participation in all or almost all labs.

B Substantial mastery of most course material. Participation in all or almost all labs.

C (above 2.0) To receive a decimal grade of 2.0 or above, you must have demonstrated sufficient mastery of the course material to, in my judgment, be capable of taking a course that has this one as a prerequisite (for example, CSS 263). It may be that your test and homework performance indicates better than 'C'-level work, but that you have chosen not to participate in labs. Such work habits are also suggestive of future success.

Assignments Assignments will be due at specific dates and times. I will *not* accept *any* lateness in this class — if your assignment is submitted late, it will *not* be graded, and you will receive a *zero* for that assignment. Except for special circumstances, such as medical and other emergencies, *no* exceptions will be made to this policy. Because there will almost always be another assignment, a test, etc. right after the due date, I believe it is better for you to submit what you have (you *should always have a partially working program*) and move on. You are more than welcome to submit work before the due date.

You will be submitting your programming assignments using the UW Catalyst "Collect It" web submission tool, linked from the course web site. Please submit only `.java` and documentation files. I will unpack your submission into a directory dedicated to it and compile and test it using a script that applies the same process to everyone's program. I will also look at your code and read your documentation. **It is your responsibility to ensure**:

- If you submit a zip or tar archive, rather than individual files, when unpacked, all of your files should be present, in the same directory as the archive (i.e., the archive should *only* contain files, not a subdirectory), and have appropriate capitalization (Unix is case-sensitive; Windows is inconsistent regarding case sensitivity.) If I need to manually rename all your files, I will reduce your program grade. I suggest that you test this by unpacking your archive in a new directory.
- That any long lines in your software are neither truncated nor wrapped.
- That the input your program expects and the output it produces **exactly** matches the specifications in the assignment.
- That your name and student number are in a comment in each source file (see coding standards, below).

I allocate credit based on your coding style, your documentation, and on your program's execution characteristics ("correctness", determined by *black-box* testing). For example, if your program does not compile, you will receive *zero* points for correctness. I will run your program against a set of test cases (which I will *not* release ahead of time — part of what everyone needs to learn is how to test code); partial credit will be awarded if it passes only some of them. I will *not* debug your program or try to figure out why it doesn't work — partial credit only comes from passing test cases. Any other way of assigning partial credit would, in my opinion, be unfair: based more on my debugging ability than the qualities of the program. Because of this, I require you to design your program before you write code, and I strongly urge you to implement your program in stages, so that you always have a partially working version. Of course, I am more than happy to meet with you about your program before or after the due date.

We will also have some written assignments. To ease homework grading and speed return of your work, please follow these homework preparation guidelines for them:

- Use lined paper with clean edges — no ragged spiral-pad "fringes," please.
- Write your name and student ID number on the upper left of the first page. Write at least your last name on each subsequent page.
- Staple your homeworks.
- Write your answers to the homework problems in order, in a single column, showing all your work. Write neatly; if I can't read it, it's wrong.
- Write your answers in the order that the questions appear. Clearly indicate the relevant question number.

Miscellaneous Hints and "Rules" The following is a brief summary of the most important things you can do to succeed in this class:

- You are responsible for making back-up copies of your work. Disk crashes, etc. are *not* acceptable reasons for extensions of assignment due dates. Note that your Windows file server directory and Linux and C&C home directories are professionally backed up.
- Assignments are due when specified. Barring illness or similar extenuating circumstances, please do not attempt to submit amendments, bug fixes, or forgotten material after the fact.
- While I do not formally require your attendance in class, I will assume that you are cognizant of everything that is covered in class, including clarifications of programming assignments, changes in due dates, etc. Material covered in class is fair game for assignments and tests, regardless of its absence from the textbooks.
- I may use email to communicate with you. It is your responsibility to ensure that email to your UW account reaches and is read by you. Note that the UW course listserve will only accept messages from addresses that are on the official list. So, if you forward your UW email, you will still need to send email messages to the listserve from your UW account. According to my experience in previous quarters, your UW email is almost certainly more reliable than any free account (e.g., Hotmail).

Special needs If you believe that you have a disability and would like academic accommodations, please contact Disability Support Services at 425.352.5307, 425.352.5303 TDD, 425.352.5455 FAX, or at dss@uwb.edu. You will need to provide documentation of your disability as part of the review process prior to receiving any accommodations.

Collaboration You are expected to do your work on your own. If you get stuck, you may discuss the problem with other students, provided that you don't copy from them. Assignments must be written up independently. You may always discuss any problem with the me or with tutors at the Quantitative Skills Center or the Writing Center. You are expected to subscribe to the highest standards of honesty.

Failure to do this constitutes plagiarism. Plagiarism includes copying assignments in part or in total, verbal dissemination of results, or using solutions from other students, solution sets, other textbooks, etc. without *crediting these sources by name*. Any student guilty of plagiarism will be subject to disciplinary action.

Problems If you have problems with anything in the course, please come and see me during office hours, or send email. I want you to succeed in this course. If you have trouble with the assignments, see me before they are due.

Design and Coding Standards

> "If builders built buildings the way programmers wrote programs, then the first woodpecker to come along would destroy civilization."
>
> "If cars had followed the same developmental path as computers, a Rolls Royce would cost $100, get a million miles per gallon, and explode once a year, killing everyone inside."

The two quotes above vividly describe the contrast between the typical practice of "programming" and that of other engineering disciplines. What is the difference? Historically, programming was not practiced as an engineering discipline. Practitioners took pride in their ability to hack out solutions. Oftimes, the more elegant solutions were, the harder they were to understand. "If it was hard to write, it should be hard to understand!" was the hacker's motto.

Much has changed in the last 20 or so years. And one of those changes has been the upgrading of Computer Science to match that of other engineering subjects. Thus, programming becomes Software Engineering, and Software Engineers spend considerable time and effort on activities other than programming. At first blush, this may seem a waste of time. However, nobody would think that of the time spent by a civil engineer designing a building, or an electrical engineer designing a computer. In those other fields, there is a big distinction made between *design* and *construction*, with the latter often not considered engineering *per se*. The same has been increasingly true of software engineering, with software design being *engineering* and programming becoming *coding*.

The reasons for this are typically couched in terms of dollars, because the largest consumers of software engineers have been corporations, and it is most convenient for them to convert everything into units of money. However, almost all of the arguments made in favor of this for the corporate environment also apply elsewhere.

The key to to understanding the advantage of the "design first" approach is to consider the entire software life cycle. When you write code without designing it ahead of time (and therefore without any design documents produced), you are making (at least) *all* of the following assumptions:

1. That only one person (yourself) will ever have to look at the code.
2. That the problem is relatively trivial (that you can keep the entire solution in your head, down to the smallest detail).
3. That the program will be used once, then thrown away (so you won't have to remember 6 months from now what you did before).
4. That there will only be one user (so no need to refer to design documentation to answer user questions).

If any of these assumptions are violated, then a design is necessary before *any* code is written (except perhaps for some prototyping, though there should still be informal designs done for those):

1. If more than one person needs to write code or work on the design, then a design document is the *only* way to communicate system function. Code is *not* documentation, it is implementation. Code does not

indicate the function a program is *supposed to* perform; it only indicates the function that a program *actually does* perform. Additionally, code is a set of formal instructions meant for a computer, it is an extremely poor way to convey meaning to human beings. Often, it is easier (and faster) to rewrite code than to understand it undocumented.

2. The solution to any nontrivial problem must be worked out in advance. Systems are often implemented in parts, and inter-operation must be assured. You may need to switch your attention from one part of the system to another, and design documentation is an essential knowledge base for storing what is known about parts you aren't currently working on.

3. Six months or more from now, it can be difficult to remember exactly why everything in the code is there. So, not only is design documentation necessary for communication with others, it is also necessary for communication with future versions of yourself.

4. Oftimes, users will ask questions about software not answered in whatever user guide is produced. At that point, if design documentation is available, an answer may be easily produced. If the answer so arrived at does not conform to the user's experience with the program, then a bug has been discovered. Therefore, design documentation also helps in the debugging process, allowing you to determine when actual system operation deviates from that which is desired.

> "A physician, a civil engineer, and a computer scientist were arguing about what was the oldest profession in the world. The physician remarked, 'Well, in the Bible, it says that God created Eve from a rib taken out of Adam. This clearly required surgery, and so I can rightly claim that mine is the oldest profession in the world.' The civil engineer interrupted, and said, 'But even earlier in the book of Genesis, it states that God created the order of the heavens and the earth out of chaos. This was the first and certainly the most spectacular application of civil engineering. Therefore, fair doctor, you are wrong: mine is the oldest profession in the world.' The computer scientist leaned back in her chair, smiled, and then said confidently, 'Ah, but who do you think created the chaos?' "
>
> — Grady Booch, *Object-Oriented Analysis and Design*

Documentation Standards

A simple approach to software development involves two parts *before* coding: determination of the desired system functionality (specification) and the actual design. The former involves major interaction with the end-users; the latter brings to bear CS knowledge (theory, algorithms, practice) and software engineering technique. We go to this trouble for one simple reason: software systems are the most complex objects routinely constructed by people. A thorough, careful design and development process is the only practical way to manage this complexity. As Grady Booch says, "We observe that this inherent complexity derives from four elements: the complexity of the problem domain, the difficulty of managing the development process, the flexibility possible through software, and the problems of characterizing the behavior of discrete systems."

Your documentation should be written so that someone else could design and code the program, or understand how your program works (including being able to modify your code).

For this class, your documentation will consist of a *specification* and a *testing report*. The specification is your way of ensuring that you understand what the assignment is asking you to do: it makes the program's functionality precise and detailed. There should be nothing ambiguous or unknown left after you write the specification. Your specification should *not* just be a regurgitation of the assignment I write; it should instead capture your understanding after all questions you may have are clarified, *before* you start designing and coding.

Divide your documentation into the following sections:

Problem statement In your own words, introduce and describe the problem to be solved. This section should also answer the following questions: What assumptions are possible? Are there special cases? Is there anything unclear in the original problem statement given to you that you clarified with me? Any assumptions that you made yourself?

Input data What is the program's input data? From where will it come (e.g., a file or the console)? In what format? How does your program know when it has reached the end of its input? What data is valid and what data is invalid? Is there an easily describable range for the data (like a range of integers)? A minimum (or first) value? A maximum (or last) value? Limits on the least (or most) amount of input the program must work on? Good answers here are necessary for development of a test plan, and there should be a clear correspondence between your description of the input and the test sets in your test plan (see below).

Output data What is the format of the output? Also, consider the questions above for input data.

Error handling What error detection and error messages are necessary? Is input validation necessary? Do you need to check/guard every I/O operation in case of failure? What about memory allocation failures? What are the warning conditions (where a message is output but program execution can continue) and error conditions (where program execution must end)?

Test Report The specifications above are the "grist" for your test plan "mill". Consider the set of possible inputs to your program (defined in the "Input Data" section). Can you break this up into subsets which are similar in some way? For example, if you were writing a tax preparation program, the part that deals with capital gains might treat negative numbers (losses) differently than positive ones (gains). For each of these subsets, choose a small number (perhaps three) of test cases (one good rule of thumb is to use the two *boundary elements* [largest and smallest values] and one *typical value*). For each input, determine what the correct output should be. The resulting table of (input, output) pairs is your *test plan*. Make sure you document the rationale for your test plan; don't just report the test plan by itself. Do *not* just produce a plan that tests erroneous input — your test plan should focus primarily on *testing the correct operation of your program.*

Use this test plan as you incrementally implement your program to check its operation. *In your documentation, indicate which tests your program passes and which it fails.*

Design and Coding Standards

You are expected to adhere to certain basic principles of good design:

Variables Each variable (whether it is a primitive type, a composite type [such as an array], or an object) has an associated *scope*. A variable's scope may be local to some small block of code (e.g., a loop), local to some function, local for each object (instance variables), or a class variable (`static`). Instance and class variables can have their accessibility modified by declaring them `private` or `public`. *You should use the most restrictive scope and access possible for all your variables, i.e., prefer local to instance, which in turn are preferred to class variables. Avoid* `public` *class members unless absolutely necessary.* You need to justify your design decisions for all class variables and all public class data members.

Methods A method should perform a *single, simply describable operation.* If you find that a "method" you are considering really does two things, then it is probably better to make it two methods.

Parameters and return values One reason for the above definition of methods is that their interfaces are kept small — they have fewer parameters and return values. Monitor each method's interface complexity. If you are passing/returning many items, this may be a sign that this is not a method.

More about methods Just like with variables, methods can have `private` or `public` accessibility. For each method, you will need to decide whether it should be publicly accessible or not. *You should make a method* `public` *only if that is truly necessary, based on the definition of the class in question.*

Classes and the implementation "wall" Classes consist of an externally-visible interface (its public members) and a hidden implementation (private members). However, a clever programmer can circumvent the wall around implementation by returning internal, implementation-dependent information. You *must not* do this — it goes against the purpose of object-oriented design. For example, you may implement a list using arrays or references; under no circumstances should you return any implementation information, such as array indices or node references.

Classes and UI/IO Classes implementing internal data types should be independent of the exact nature of any particular program. *Such classes should not include any user interface operations.* As a simple example, imagine you are designing a vector class, and that you included operations tied to a graphical user interface. This would mean that your class would be *unusable* in a non-GUI environment, such as a computer controlling a car's engine. You should detach issues of user interface and I/O from such class design — they are *separate* parts of the design. Implement I/O in classes dedicated to that purpose.

Coding standards means writing code that is easily understood and including comments that clearly document its function. Code clarity is aided by consistent and useful indentation, identifiers with descriptive names and naming conventions, and the use of language constructs like `final`. More precisely, our course coding standards are:

Formatting Blocks of code should be indented *three spaces*. This includes the bodies of functions. If you use an IDE, make sure it actually writes space characters, not tab characters, into the source files. Limit line length to 80 characters; do not assume that someone reading your code will have a gigantic monitor or good enough eyesight to set a small enough font size to make code with very long lines readable. Assume that the reader will examine your code as plain text files, not using an IDE.

Variables Variables should be given descriptive names, unless they are very clearly just loop counters or the like. There should be comments associated with each variable declaration explaining how the variable fits into the algorithm, and including invariant information such as its legal range of values.

File comments Each file should begin with a comment containing the file name, author name, date, and a description of the purpose of the code it contains. The file that contains `main()` should also include documentation for the overall program: a description of the program's input and output, how to use the program, assumptions such as the type of data expected, exceptions, and a brief description of the major algorithms and key variables. This is the information you generated in your design, *before* you started coding. It is expected that you will merely copy the appropriate sections of the your documentation into comments for each file and function (see method comments, below).

Method comments Each method should be preceded by a comment with a short description of its purpose, arguments, and return values. You are encouraged to use javadoc to format your comments; there is a similar syntax, used by a program called doxygen, for other programming languages.

Tentative Course Schedule

Date	Topics	Readings	Assignments
3/30	Welcome; let's get started!	Savitch, § 4.1	Program 1 assigned
4/1	Happy April Fool's Day!; Encapsulation; Abstract data types	Savitch, § 4.2	
4/6	Overloading; Constructors	Savitch, § 4.3-4.4; Ermann, ch. 9	
4/8	Propositional logic	Rosen, § 1.1-1.4	Program 1 due; Written HW 1 assigned
4/13	References; Class parameters	Savitch, § 5.1-5.2	Written HW 1 due; Program 2 assigned
4/15	Tax Day; References, cont'd; packages	Savitch, § 5.3-5.4; Ermann, ch. 16	Written HW 2 assigned
4/20	Number bases	Rosen, § 3.6	Program 2 due; Program 3 assigned
4/22	Earth Day; Sets and functions	Rosen, § 2.1-2.3	Written HW 2 due; Written HW 3 assigned
4/27	Inheritance; Exceptions	Savitch, § 7.1, 9.1-9.2	Program 3 due; Program 4 assigned
4/29	Recursion	Savitch, § 11.1-11.2; Ermann, ch. 11	Written HW 3 due
5/4	Midterm		
5/6	Recursion, cont'd	Savitch, § 11.3	Program 4 due; Program 5 assigned
5/11	Introduction to algorithm analysis	Rosen, § 3.1-3.3	Written HW 4 assigned
5/13	Anyone remember Camp Casey?; Search algorithms		Program 5 due; Program 6 assigned
5/18	Interfaces; `Arraylist`	Savitch, § 13.1-13.2, 14.1	Written HW 4 due
5/20	Data Structures: stacks and queues	Savitch, § 15.1, 15.4; Ermann, ch. 19	Program 6 due; Program 7 assigned
5/25	Memorial Day		
5/27	Collections; generics	Savitch, § 14.2, 16.1-16.2	Program 7 due; Program 8 assigned
6/1	Introduction to sorting algorithms	Savitch, § 12.2	
6/3	Wrap-up and review for final		Program 8 due
6/8	Final exam (at usual class time); Have a great summer!		