Revised 10/10/'12

CS 394 - F01
Software Construction – 3 credits
Spring 2013

Instructor: Dr. Chris Hartman
Email: cmhartman@alaska.edu
Office: 201-D Chapman
Office Phone: 474-5829
Office Hours: TBD or by appointment

Prerequisites: CS 311

Text: *Practical Tools and Techniques for Software Development* by Edward Crookshanks, CreateSpace Independent Publishing Platform; 2nd edition (April 3, 2012)

Course BlackBoard site at http://classes.uaf.edu

Schedule: TBD
Location and Time: TBD

Assessment of the following items will be used in the following proportions to determine student grades.

Assignments	30%
Group Projects	40%
Class Participation	20%
Final Exam	10%

Course description:
From the catalog: CS F394 Software Construction
Methods for programming and construction complete computer applications, including refactoring, performance measurement, process documentation, unit testing, version control, integrated development environments, debugging and debuggers, interpreting requirements, and design patterns. Prerequisite: CS 311. (3+0)

This is a trial course for Spring 2013 which will end up being a required course for all Computer Science students, leading up to the senior capstone sequence of 471/472. In this course we will learn several techniques (see catalog description) for writing large-scale programs that lead to better software with fewer bugs.

The textbook cites the reasons for learning these topics as follows:

The purpose of this companion guide is to discuss and provide additional resources for topics and technologies that current university curriculums may leave out. Some programs or professors may touch on some of these topics as part of a class, but individually they are mostly not worthy of a dedicated class, and collectively they encompass some of the tools and practices that should be used throughout a software developer's career. Use of these tools and topics is not mandatory, but applying them will give the student a better understanding of the practical side of software development.
In addition, several of these tools and topics are the "extra" goodies that employers look for experience working with or having a basic understanding of. In discussions with industry hiring managers and technology recruiters, the author has been told repeatedly that fresh college graduates, while having the theoretical knowledge to be hired, often times are lacking in more practical areas such as version control systems, unit testing skills, debugging techniques, interpreting business requirements, and others. This is

not to slight or degrade institutional instruction, only to point out that there are tools and techniques that are part of enterprise software development that do not fit well within the confines of an educational environment. Knowledge of these can give a student an advantage over those who are unfamiliar with them. This guide will discuss those topics and many more in an attempt to fill in the practical gaps. In some cases the topics are code-heavy, in other cases the discussion is largely a survey of methods or a discussion of theory. Students who have followed this guide should have the means to talk intelligently on these topics and this will hopefully translate to an advantage in the area of job hunting. While it would be impossible to cover all tools and technologies, the ones covered in this guide are a good representative sample of what is used in the industry today. Beyond the theoretical aspects of computer science are the practical aspects of the actual implementation in an enterprise environment; it is this realm that this book attempts to de-mystify. In short, it is hoped that this companion guide will help graduates overcome the "lack of practical experience" issue by becoming more familiar with industry standard practices and common tools. In this volume we cannot create experts, but at least provide enough cursory knowledge such that the reader can discuss the basics of each topic during an interview. With a little practice and exploration on their own, the student should realize that supplementing an excellent theoretical education with practical techniques will hopefully prove useful not only in writing better software while in school, but also translate to an advantage when out of school and searching for a job.

You are expected to be proficient in the material from CS 311 (a pre-requisite) such as advanced C++ programming, common data structures and algorithms, and beginning software engineering techniques.

Expected Student Outcomes:

Ability to measure actual performance on a given architecture
Ability to implement a software system
Ability to design a large software system (as a group)
Ability to create software process documents while following a defined process (as a group)
Ability to create effective software process documents
Ability to write code without bugs
Ability to effectively use a version control system to develop software
Experience in using design patterns to plan software architecture
Experience with code reviews

Instructional Methods – Classroom lectures, discussion of external readings and case studies and in-class code review, group presentations.

Class Participation – You will be expected to participate in discussions of the reading material and case studies, and to actively participate in code-review sessions. Approximately 1/3 of classroom time will be spent on these activities.

Group Projects – You'll complete three small software development projects, each of which goes all the way from specification and design to coding and testing. There will be one project with groups of two, one project with groups of three, and one project with larger groups. You'll have the opportunity to choose whom to work with. Any pair of students, however, will only be allowed to work together on a single project. Collaboration is encouraged, although each team member is required to participate roughly equally in every activity (design, implementation, test, documentation, presentation), and I may ask for an accounting of what each team member did. Each project will have multiple due dates, with different deliverables. On the preliminary due dates you will turn documents having to do with exploring the problem and initial design decisions. On the final date, you will turn in final design decisions and working code. Code and design documents will be handed in electronically (by committing in the repository before the deadline). Each project will have at least two in-class presentations (discussing design decisions and finally demonstrating working code.) While each team member will not be required to participate in each presentation, each team member must at least give some part of some presentation.

Assignments – Assignments will be required generally on a weekly to biweekly basis. The assignments will reinforce lecture concepts, introduce material needed for group projects, and demonstrate application of critical thinking skills. Unless otherwise specified, all assignments must be done on an individual basis.

Policies – Examinations **must** be taken at the scheduled time. In particular, there **will be no** early final exams. You may discuss homework and programming assignments with others, but everything you turn in **must** be your own work.

Disabilities Services – The Office of Disability Services implements the Americans with Disabilities Act (ADA), and insures that UAF students have equal access to the campus and course materials. I will work with the Office of Disabilities Services to provide reasonable accommodation to students with disabilities.

Extremely Tentative Schedule:

Week 1: Version control - tools and purpose.

- Theory
- Resolving conflicts
- Software: Subversion (svn), Git, CVS
- What to keep in a repository

Weeks 2-3: Introduction to the Software Development Process

- Requirements
- Specification
- Architecture
- Design
- Implementation
- Testing
- Debugging
- Deployment
- Maintenance
- Introduction to design patterns
- Modeling languages (UML, etc.)

Week 3: Unit Testing and Test Driven Development

- Theory
- Frameworks
- Automated testing
- Static analysis
- Examples
- Tools for TDD

Week 4: Introduction to Requirements, Project 1

- Stakeholder identification
- Business requirements
- Functional requirements
- Measurable goals
- Prototypes
- Use cases
- Specification
- Project 1 assignment and discussion

Week 5: Build tools, automated build engineering, and continuous integration. The code review process. Project 1 continued.

- Build tools, etc.
  - Make
  - Ant
  - NAnt/MSBuild
  - Maven
  - Continuous Integration tools
  - Examples
- Code review
  - Peer review
  - Automated code review
  - More?
- Project 1 design presentations
- Project 1 work time

Week 6: Debugging and Profiling, Project 1 completion

- Debugging
    - Breakpoints
    - Stepping
    - Stack trace
    - Logging
- Performance measurement/profiling/optimizing

Week 7: Refactoring - purpose and automated tools. Project 2

- See Wikipedia's list of refactoring techniques
- Project 2 assignment and discussion

Week 8: Design patterns, Project 2 continued

- Introduction to design patterns

Week 13: Documentation and Software Process documents

Week 14: Comparison of development methodologies (waterfall and agile).

Project 2
Tentative rough outline, not for class distribution

Three Person Groups

Based on http://web.mit.edu/6.005/www/sp11/projects/draw_blank/assignment.html

Design and implement a "C++ graphics language" to PostScript translator.

By "graphics language" we mean a base class for a Shape, and derived classes for basic shapes such as Circle, Square, etc. Also, derived classes for compound shapes (shapes composed of other shapes) such as RotatedShape, ScaledShape, VerticalStack (set of other shapes arranged vertically), OverlappingStack (set of other shapes drawn overlapping) etc. Clean up this paragraph and give more formal descriptions of all required shapes.

Shapes will have bounding boxes with width and height. It is up to you how to implement the rest.

For the translator part, you will need a method for your Shape class, that when called, generates a sequence of PostScript commands. The client will specify the name of the resulting file, which will be a legal PostScript file that can be previewed on screen or printed on paper.

Tasks:

1. Become familiar with PostScript (from an assignment maybe?)
2. Design the shape language (expressed as definitions of a base class and several derived classes, member functions must be declared but need not be defined yet.)
3. Design the translation part of your code, expressing your design with the tools discussed in class such as UML diagrams and outlines of the methods. Explain the advantages and disadvantages of your design by comparing it to at least one alternative, fleshing out the alternative with UML diagrams and outlines of the methods to the extent necessary to make your points clear and easily understood.
4. Extend your design by designing three or more nice shapes of your own. For example, you might implement a class Skylines that contains methods returning randomized skylines with the number/height/shape of buildings specified as input arguments to the methods. Or you might implement a class Fractals that creates some fractal figures with a recursion limit specified by the user. These are just examples; it's entirely up to you. A prize will be awarded for the best shape. [This text cribbed directly from the MIT assignment, think up my own examples and rewrite.]
5. Present your design in class. Your presentation must include design choices along with justification, description of your planned implementation, and your planned extensions.
6. Implement your design, using proper TDD standards.
7. Demonstrate your working software with an in-class demo.
8. Reflect. Write a brief commentary saying what you learned from this experience. What was easy? What was hard? What was unexpected? Briefly evaluate your solution, pointing out its key merits and deficiencies. Was your design flexible and modular enough to allow easy inclusion of more features? How could you make it more flexible? Critique the specification of the shape language. Is the language expressive enough to create interesting drawings? Are the shape abstractions well-designed, or not? Do they make certain drawings easy/difficult to create? If you were to design the shape language from scratch, what would you do differently?

Deliverables:
There are two deadlines for this project. On date you will hand in a design document that includes your shape language (task 2) and your design analysis (task 3). Class presentations for task 5 will be scheduled during class on date. On date you will turn in your implementation (with test suite), your

design extensions (task 4), and your reflections (task 8.) Class demonstrations of your working software will be scheduled during class on date.

Grading:
90% of your project grade will be allotted to the design and implementation (including extensions), and 10% to the example shapes you created. Of the 90%, 30% will be allotted to the design of your recursive types and your design critique, 40% to the code (half for structure and half for correctness), and 20% to the testing strategy, test cases and reflections. [Not sure on these percentages yet.]

Hints:
1. When implementing a visitor that generates a PostScript program, it may be useful to (1) assume, on entry to each visiting method, that the point around which the shape is to be centered has already been correctly set; and (2) for basic shapes, to draw the shape by starting a new path, drawing, closing the path, and calling stroke; and (3) for some visitor methods, to use gsave on entry and grestore before exit. These are only suggestions, however, and you might find a different (and maybe better) approach.
2. The width and height of the bounding box for a polygon with n sides of length e is given by the following formulas:
    - Case 1: n is odd.
      height = $e(1+\cos(\pi/n))/(2\sin(\pi/n))$
      width = $(e \sin(\pi(n-1)/2n))/(\sin(\pi/n))$
    - Case 2: n is divisible by 4.
      height = $e(\cos(\pi/n))/(\sin(\pi/n))$
      width = $(e \cos(\pi/n))/(\sin(\pi/n))$
    - Case 3: n is divisible by 2, but not by 4.
      height = $e(\cos(\pi/n))/(\sin(\pi/n))$
      width = $e/(\sin(\pi/n))$
3. In PostScript, use showpage to finally draw the current page and create a new page.

Outside resources:

Design patterns:

- Here is a good software development wiki (although it can be hard to navigate). It has some useful content on Design Patterns. If you're looking for information on a particular design pattern see the Software Design Patterns Index.
- I also like the SourceMaking page on design patterns.

TDD:

- Uncle Bob's blog or maybe at http://blog.objectmentor.com/ (Software Best Practices)
- Introduction to Test Driven Development

Agile Software Development

- http://agilemanifesto.org/
- http://blogs.forrester.com/mike_gualtieri/11-10-12-agile_software_is_a_cop_out_heres_whats_next
- `http://www.agilemodeling.com/essays/singleSourceInformation.htm` and others from here.
-

General Concepts

- Dependency Inversion Principle
- Other software engineering principles

Static Analysis

- http://www.altdevblogaday.com/2011/12/24/static-code-analysis/

Recommended books (Amazon paperback price/Kindle price):

Strongly:

Head First Design Patterns, $28/$25
Clean Code, $39/$23
Agile Software Development , Principles, Patterns, and Practices $55 or $35 used/NA

Also recommended:

Code Complete, $28/$20
Why Programs Fail: A Guide to Systematic Debugging, $55/$40

Other: (Need to find review copies of these and see if they're any good)

The Pragmatic Programmer

Thoughts on assignments:
Lots of reccomended reading of short blog posts, etc. Some way to check that they've actually read it?

Possible assignments:

- Simple introduction to Eclipse and Unit testing
- Introduction to postscript (in prep for project 1)
- Bowling score calculator (TDD example)
- Calculator (GUI?)
- Web page backed by database, using SQL (possible project?)
-

All assignments turned in using version control, probably Git?

Similar courses

- MIT has offered a similar course (about 60% overlap) every semester since 2007, available here: https://stellar.mit.edu/S/course/6/sp08/6.005/ All years from 2007 to 2012, see sample projects etc.
- Sample projects from 2007 MIT course: http://stellar.mit.edu/S/course/6/fa07/6.005/materials.html#topic4
- Also (same course) available from MIT Open Courseware: http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-005-elements-of-software-construction-fall-2011/lecture-notes/
- https://sewiki.iai.uni-bonn.de/teaching/lectures/atsc/2012/start

Require team contracts as in http://web.mit.edu/6.005/www/sp12/projects/project1/teamcontract.html for projects?

Possible additional material

- Segment on smart pointers
- Segment on networking? Or does this still belong on OS?
- Segment on Boost
- Segment on "clean code"

Require (as part of class participation) at least three links to relevant blog posts or news articles?

Need information from Genetti on software process documents ("**D8 - Ability to create software process documents while following a defined process (as a group)" and "F4 - Ability to create effective software process documents**")

What do we mean by "cover test planning" to help with **D2 - Ability to design a large software system (as a group)?**

Add several from http://en.wikipedia.org/wiki/List_of_software_development_philosophies to software engineering section.

Make sure all design patterns are covered in those lists.

Figure out Database/SQL/Web project

In general (not software construction course) where do students see linux?