V22.0474-001 Software Engineering
Spring 2008

Lecture 1

Clark Barrett, New York University

Course Information

Prerequisites

Familiarity with C++, experience with at least one large software project.

Web Page

http://cs.nyu.edu/courses/spring08/V22.0474-001/index.html

Email List

http://cs.nyu.edu/mailman/listinfo/v22_0474_001_sp08

Books

Recommended: Sommerville, Ian. Software Engineering, 8th Edition.
Recommended: Meyers, Scott. Effective C++, Third Edition.
Recommended: McConnell, Steve. Code Complete, Second Edition.
I will be drawing material from many sources. These sources will be listed at the beginning of each lecture. All sources are recommended reading material.

Course Information

Exams

There will be a midterm, but no final.

Assignments

There will be a weekly assignment which will generally be due the following week (unless otherwise indicated). Assignments will alternate between exercises to solidify concepts from the lecture material and tasks relating to the semester-long software project. Assignments will be posted on the class web page.

Project

The bulk of your grade will be determined by your performance on a semester-long software project. The project will be done in teams. Throughout the semester, various project tasks will be assigned. Most of these will require you to produce and maintain design documents. Some of them will also require you to prepare a presentation to be given in class. Each group will have the opportunity to review some of the documents produced by one of the other groups.

Course Information

Grading

Project Proposal: 10%
Individual Presentation: 10%
Weekly Assignments: 30%
Midterm: 20%
Final Project: 30%

Each of you will give feedback on the performance of the other members of your group. *This feedback WILL affect your grade.*

Academic Integrity

Please review the departmental academic integrity policy.

In this course, you are encouraged to work together on the project and assignments. However, any help you receive *must* be clearly noted on your assignment. Also, you should consult the instructor before using materials or code other than that provided in class. Copying code or other work without giving appropriate acknowledgment is a serious offense with consequences ranging from no credit to potential expulsion.

Course Information

Acknowledgments

In addition to the books mentioned above, I will draw from a number of other sources throughout the semester. Each lecture will include a list of source material. In addition, I have adapted many ideas (with permission) from a course developed by George Necula and Alex Aiken at Berkeley.

Sources for today's lecture:

Necula, George and Aiken, Alex. *CS 169: Software Engineering*. Lecture notes, Fall 2004.

McConnell, Steve. *Code Complete*, Second Edition (abbreviated as CC2).

Sommerville, Ian. *Software Engineering*, Eighth Edition, chapters 1–3 (abbreviated as SE).

What is Software Engineering?

What is Software Engineering?

The art of programming has taken 50 years of continual refinement to reach this stage. By the time it reached 25, the difficulties of building big software loomed so large that in the autumn of 1968 the NATO Science Committee convened some 50 top programmers, computer scientists and captains of industry to plot a course out of what had come to be known as the software crisis. Although the experts could not contrive a road map to guide the industry toward firmer ground, they did coin a name for that distant goal: software engineering, now defined formally as *"the application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software."*

A quarter of a century later software engineering remains a term of aspiration.

-From "Software's Chronic Crisis," in *Scientific American*, Sept. 1994.

What is Software Engineering?

From IEEE Standard 610.12:

> *"The application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software."*

- This is a pretty good definition
- But it says *what* software engineering should accomplish, not *how*
- People disagree on the best ways to accomplish this goal
- We know a lot more than we used to, but

  Software engineering remains a term of aspiration.

Why do we need Software Engineering?

Why do we need Software Engineering?

Creating quality software is hard and fraught with peril.

Software Construction Pitfalls: Management

We have books with rules. Isn't that all we need?

People have different books and different rules.

If we fall behind, we add more programmers.

Brooks' Law: *"Adding manpower to a late software project makes it later"* (from *The Mythical Man-Month*).

We can outsource it.

If you cannot manage and control it internally, you will have difficulty doing it with outsiders.

Software Construction Pitfalls: Customer

We can refine the requirements later

A recipe for disaster

The good thing about software is that we can easily change it later

As time passes, the cost of changes grows rapidly.

Software Construction Pitfalls: Practitioner

Let's write the code, so we'll be done faster

- *"The sooner you start to code, the longer the program will take."* Roy Carlson, University of Wisconsin (from *More Programming Pearls* by John Bentley).
- 60-90% of the development effort comes *after* the initial release (CC2, p. 13).

Until I finish it, I cannot assess its quality

Software and design reviews are more effective than testing.

There is no time for software engineering

Do you have time to do it again?

How do we apply Software Engineering?

How do we apply Software Engineering?

So...how do we apply a systematic, disciplined, and quantifiable approach to the development, operation and maintenance of software?

How do we apply Software Engineering?

So...how do we apply a systematic, disciplined, and quantifiable approach to the development, operation and maintenance of software?

Look for metaphors in other tasks we understand well.

Metaphors for Creating Software

How do we apply Software Engineering?

So...how do we apply a systematic, disciplined, and quantifiable approach to the development, operation and maintenance of software?

Look for metaphors in other tasks we understand well.

Metaphors for Creating Software

- Software Penmanship: Writing Code

How do we apply Software Engineering?

So...how do we apply a systematic, disciplined, and quantifiable approach to the development, operation and maintenance of software?

Look for metaphors in other tasks we understand well.

Metaphors for Creating Software

- Software Penmanship: Writing Code

  Breaks down with more than one author.

How do we apply Software Engineering?

So...how do we apply a systematic, disciplined, and quantifiable approach to the development, operation and maintenance of software?

Look for metaphors in other tasks we understand well.

Metaphors for Creating Software

- Software Penmanship: Writing Code

  Breaks down with more than one author.

- Software Farming: Growing a System

How do we apply Software Engineering?

So...how do we apply a systematic, disciplined, and quantifiable approach to the development, operation and maintenance of software?

Look for metaphors in other tasks we understand well.

Metaphors for Creating Software

- Software Penmanship: Writing Code

  Breaks down with more than one author.

- Software Farming: Growing a System

  Captures incremental process, but implies that software somehow grows by itself.

How do we apply Software Engineering?

So...how do we apply a systematic, disciplined, and quantifiable approach to the development, operation and maintenance of software?

Look for metaphors in other tasks we understand well.

Metaphors for Creating Software

- Software Penmanship: Writing Code

  Breaks down with more than one author.

- Software Farming: Growing a System

  Captures incremental process, but implies that software somehow grows by itself.

- Software Oyster Farming: System Accretion

How do we apply Software Engineering?

So...how do we apply a systematic, disciplined, and quantifiable approach to the development, operation and maintenance of software?

Look for metaphors in other tasks we understand well.

Metaphors for Creating Software

- Software Penmanship: Writing Code

  Breaks down with more than one author.

- Software Farming: Growing a System

  Captures incremental process, but implies that software somehow grows by itself.

- Software Oyster Farming: System Accretion

  Software grows layer by layer over time, like a pearl from a grain of sand.

How do we apply Software Engineering?

So...how do we apply a systematic, disciplined, and quantifiable approach to the development, operation and maintenance of software?

Look for metaphors in other tasks we understand well.

Metaphors for Creating Software

- Software Penmanship: Writing Code

  Breaks down with more than one author.

- Software Farming: Growing a System

  Captures incremental process, but implies that software somehow grows by itself.

- Software Oyster Farming: System Accretion

  Software grows layer by layer over time, like a pearl from a grain of sand.

- Software Construction: Building Software

How do we apply Software Engineering?

So...how do we apply a systematic, disciplined, and quantifiable approach to the development, operation and maintenance of software?

Look for metaphors in other tasks we understand well.

Metaphors for Creating Software

- Software Penmanship: Writing Code

  Breaks down with more than one author.

- Software Farming: Growing a System

  Captures incremental process, but implies that software somehow grows by itself.

- Software Oyster Farming: System Accretion

  Software grows layer by layer over time, like a pearl from a grain of sand.

- Software Construction: Building Software

  Powerful metaphor that is very helpful when talking about software engineering.

How do we apply Software Engineering?

The Construction Metaphor

- Ad hoc methods work for small projects, but not for large: a dog house vs. a skyscraper
- Labor, not materials, is the main expense: moving a wall vs. changing requirements
- Don’t build from scratch what you can get pre-built.
- Careful planning benefits the project.
- Extremely large projects have to be “over-engineered”.

The Construction Metaphor

Of course, no metaphor is perfect:

In software construction, it is often more difficult to reuse components

> *Computing is the only profession in which a single mind is obliged to span the distance from a bit to a few hundred megabytes, or nine orders of magnitude.* Steve McConnell, *Code Complete*.

Physics guides physical construction

> *Einstein argued that there must be a simple explanation of nature, because God is not capricious or arbitrary. No such faith comforts the software engineer.* Fred Brooks, Jr. (author of *The Mythical Man-Month*).

Software evolves and changes much more than a typical physical building

> Software engineering must address change and evolution as an essential part of the process.

Contents of the Course

The topics we will cover in this class fall into the following broad categories:

Software Processes

- Strategies for managing the overall construction process

Requirements

- Understanding the goals of the software project. This also helps us evaluate whether the project was successful.

Configuration Management

- Managing components and change

Design and Implementation

- The actual construction of the code.

Verification and Validation

- Assessing the quality and correctness of software

Code Optimization and Tuning

- Finding and improving weak points in the code.

The Project

Constraints

- Coded (primarily) in C++ (more on this later)
- User-interface may use non-C++ code if desired
- Must compile and run using GNU UNIX tools (we will cover the basics in class)

A BIG project

- Can be (almost) anything
- Must have a real person who is the customer

Done in teams

- You do everything
- Design, code, and test in eight assignments

Be prepared for a lot of work (and fun)

The Project

Good software engineering can be learned

- It is hard to teach
- Most people learn through experience (i.e. mistakes)

This class aims to provide some of that experience

- Do a project as part of a team
- Listen to the presentations of other teams
- Present your own progress to the class (each team will present 4 times)

The Project

Eight Project Stages

- Project nominations
- Project selection and team assignments
- Requirements and specification
- Project design and plan
- Design review (done by another team)
- Revised design and plan
- Validation and Testing (done by another team)
- Final Report

Project nominations

Your first assignment is to propose a project:

- 1-2 page proposal
- Must have a customer
- Due on Thursday, January 24
- Details on web page

Why C++?

Why not JAVA?

- In the “real world”, more large projects are done in C++ than Java.
- NYU focuses on Java, but every self-respecting CS graduate should know C++. I want to help you respect yourself.

It’s easier to screw up in C++.

- So you have to know what you’re doing.
- Sometimes it’s good to be forced to know what you’re doing.

C++ is generally considered more mature and more efficient

- Sometimes this makes a difference.

Finally...

- I have more experience with C++, so I can teach you more.
- Hopefully, most of the concepts you learn here will be independent of the programming language.

C++

What to do if you don't know C++ very well

- Give up.
- Catch up.

Some C++ Resources

- C++ for Java Programmers: http://www.cs.wisc.edu/∼hasti/cs368/CppTutorial/
- C++ language tutorial: http://www.cplusplus.com/doc/tutorial/
- Lippman, Stanley. *Essential C++*.
- Lippman, Stanley. *C++ Primer*.

Getting Started

Log onto a unix machine

- Install cygwin for Windows
- Install linux on your laptop
- Use a terminal window on a Mac
- Use the CIMS machines

Getting Started

Start the emacs editor

```
> emacs hello.cpp
```

If that doesn't work, try

```
> emacs -nw hello.cpp
```

Write some code

```
#include <iostream>

using namespace std;

int main(void)
{
  cout << "Hello world\n";
  return 0;
}
```

Getting Started

Exit emacs by typing Ctrl-x Ctrl-c y

Compile

```
> g++ -o hello hello.cpp
```

Run it

```
> ./hello
```

Hello world

Repeat and generalize

More resources:

- GNU Emacs page: http://www.gnu.org/software/emacs/
- Commonly used UNIX commands: http://infohost.nmt.edu/tcc/help/unix/unix_cmd.html
- g++ tutorial: http://www.wm.edu/computerscience/computing/gpp.php