

Introduction and Course Overview

6.170 Lecture 1

Spring 2004

1 Why Software Engineering Matters

Software's contribution to US economy is very significant. In 1999, US software companies had \$141 *billion* in sales. Software has the greatest trade surplus of exports, including agriculture, aerospace, and chemicals. In 1996, the US exported \$24B worth of software, and imported \$4B for a \$20B surplus. In contrast, for manufactured goods, the numbers were \$200B and \$264B. (from *Software Conspiracy*, Mark Minasi, McGraw Hill, 2000).

Software's role in infrastructure is increasing:

- not just the Internet
- transportation, energy, medicine, finance

Software is becoming pervasive in embedded devices. New cars, for example, have between 10 and 100 processors for managing all kinds of functions from music to braking.

How good is our software?

- failed developments
- accidents
- poor quality software

1.1 Development failures

IBM survey, 1994

- 55% of systems cost more than expected
- 68% overran schedules
- 88% had to be substantially redesigned

Advanced Automation System (FAA, 1982-1994)

- industry average was \$100/line, expected to pay \$500/line
- ended up paying \$700-900/line
- \$6B worth of work discarded

Bureau of Labor Statistics (1997)

- for every 6 new systems put into operation, 2 cancelled
- probability of cancellation is about 50% for biggest systems
- average project overshoots schedule by 50%
- 3/4 systems are regarded as 'operating failures'

1.2 Accidents

"The most likely way for the world to be destroyed, most experts agree, is by accident. That's where we come in. We're computer professionals. We cause accidents."

Nathaniel Borenstein, inventor of MIME, in: *Programming as if People Mattered: Friendly Programs, Software Engineering and Other Noble Delusions*, Princeton University Press, Princeton, NJ, 1991.

Therac-25 (1985-87)

- radiotherapy machine with software controller
- hardware interlock removed, but software had no interlock
- software failed to maintain essential invariants: either electron beam mode or stronger beam and plate intervening, to generate X-rays
- several deaths due to burning, patients received > 10,000 rads instead of typical 200 rad dose
- programmer had no experience with concurrent programming
- see: <http://sunnyday.mit.edu/therac-25.html>

You might think that we'd learn from this and such a disaster would never happen again. But...

- International Atomic Energy Agency declared 'radiological emergency' in Panama on 22 May, 2001
- 28 patients overexposed; 9 died, 5 as a result of exposure: many of the remaining expected to develop 'serious complications that may ultimately prove fatal'
- Experts found radiotherapy equipment 'working properly'; cause of emergency lay with data entry
- If data entered for several shielding blocks in one batch, incorrect dose computed, but the printout showed the doses as if the data were interpreted correctly
- FDA, at least, concluded that 'interpretation of beam block data by software' was a factor
- see <http://www.fda.gov/cdrh/ocd/panamaradexp.html>

Mars Climate Orbiter (1999)

- approached too close to Mars (60 km; should have been 150 km)
- lost, presumably burned up
- the miscalculation was traced to a discrepancy of units: one programming team assumed metric units (kilograms/sec), but another team used English (pounds/sec)

Mars Spirit rover (2004)

- stopped working properly several days after landing
- Flash memory filled with too many files (created by takeoff, cruise, and landing, but never cleaned up)

In the short term, these problems will become worse because of the pervasive use of software in our civic infrastructure. A PITAC report recognized this, and has successfully argued for increase in funding for software research:

“The demand for software has grown far faster than our ability to produce it. Furthermore, the Nation needs software that is far more usable, reliable, and powerful than what is being produced today. We have become dangerously dependent on large software systems whose behavior is not well understood and which often fail in unpredicted ways.”

Information Technology Research: Investing in Our Future

President’s Information Technology Advisory Committee (PITAC)

Report to the President, February 24, 1999

Available at <http://www.ccic.gov/ac/report/>

1.3 Software Quality

One measure: bugs/kloc

- measured after delivery
- industry average is about 10
- high quality: 1 or less

Praxis CDIS system (1993)

- UK air-traffic control system for terminal area
- used precise spec language, very similar to the object models we’ll learn
- no increase in net cost
- much lower bug rate: about 0.75 defects/kloc
- even offered warranty to client!

Of course, quality isn’t just about bugs. You can test software and eliminate most of the bugs that cause it crash, but end up with a program that’s impossible to use and fails much of the time to do what you expect, because it has so many special cases. To address this problem, you need to build quality in from the start.

2 Think Before You Code

“You know what’s needed before we get good software? Cars in this country got better when Japan showed us that cars could be built better. Someone will have to show the industry that software can be built better.”

John Murray, FDA’s software quality guru

quoted in *Software Conspiracy*, Mark Minasi, McGraw Hill, 2000

That’s you!

Our aim in 6170 is to show you that ‘hacking code’ isn’t all there is to building software. In fact, it’s only a small part of it. Don’t think of code as part of the solution; often it’s part of the problem. We need better ways to talk about software than code, that are less cumbersome, more direct, and less tied to technology that will rapidly become obsolete.

Design is important because:

- thinking in advance always helps (and it’s cheap!)
- can’t add quality at the end: contrast with reliance on testing; more effective, much cheaper
- makes delegation and teamwork possible
- design flaws affect user: incoherent, inflexible and hard to use software
- design flaws affect developer: poor interfaces, bugs multiply, hard to add new features

It’s a funny thing that computer science students are often resistant to the idea of software development as an engineering enterprise. Perhaps they think that engineering techniques will take away the mystique, or not fit with their inherent hacker talents. On the contrary, the techniques you learn in 6170 will allow you to leverage the talent you have much more effectively.

2.1 The Netscape Story

For PC software, there’s a myth that design is unimportant because time-to-market is all that matters. Netscape’s demise is a story worth pondering in this respect.

The original NCSA Mosaic team at the University of Illinois built the first widely used browser, but they did a quick and dirty job. They founded Netscape, and between April and December 1994 built Navigator 1.0. It ran on 3 platforms, and soon became the dominant browser on Windows, Unix and Mac. Microsoft began developing Internet Explorer 1.0 in October 1994, and shipped it with Windows 95 in August 1995.

In Netscape’s rapid growth period, from 1995 to 1997, the developers worked hard to ship new products with new features, and gave little time to design. Most companies in the shrink-wrap software business (still) believe that design can be postponed: that once you have market share and a compelling feature set, you can ‘refactor’ the code and obtain the benefits of clean design. Netscape was no exception, and its engineers were probably more talented than many.

Meanwhile, Microsoft had realized the need to build on solid designs. It built NT from scratch, and restructured the Office suite to use shared components. It did hurry to market with IE to catch up with Netscape, but then it took time to restructure IE 3.0. This restructuring of IE is now seen within Microsoft as the key decision that helped them close the gap with Netscape.

Netscape’s development just grew and grew. By Communicator 4.0, there were 120 developers (from 10 initially) and 3 million lines of code (up a factor of 30). Michael Toy, release manager, said:

“We’re in a really bad situation ... We should have stopped shipping this code a year ago. It’s dead... This is like the rude awakening... We’re paying the price for going fast.”

Interestingly, the argument for modular design within Netscape in 1997 came from a desire to go back to developing in small teams. Without clean and simple interfaces, it’s impossible to divide up the work into parts that are independent of one another.

Netscape set aside 2 months to re-architect the browser, but it wasn’t long enough. So they decided to start again from scratch, with Communicator 6.0. But 6.0 was never completed, and its developers were reassigned to 4.0. The 5.0 version, Mozilla, was made available as open source, but that didn’t help: nobody wanted to work on spaghetti code.

In the end, Microsoft won the browser war, and AOL acquired Netscape. Of course this is not the entire story of how Microsoft's browser came to dominate Netscape's. Microsoft's business practices didn't help Netscape. And platform independence was a big issue right from the start; Navigator ran on Windows, Mac and Unix from version 1.0, and Netscape worked hard to maintain as much platform independence in their code as possible. They even planned to go to a pure Java version ('Javagator'), and built a lot of their own Java tools (because Sun's tools weren't ready). But in 1998 they gave up. Still, Communicator 4.0 contains about 1.2 million lines of Java.

This section is excerpted from an excellent book about Netscape and its business and technical strategies. You can read the whole story there:

Michael A. Cusumano and David B. Yoffie. *Competing on Internet Time: Lessons from Netscape and its Battle with Microsoft*, Free Press, 1998. See especially Chapter 4, Design Strategy.

Note, by the way, that it took Netscape more than 2 years to discover the importance of design. Don't be surprised if you're not entirely convinced after one term; some things come only with experience.

3 About 6.170

This course is actually three courses in one:

- A crash course in object-oriented programming,
- Software design in the medium, and
- A studio course on team construction of software.

We emphasize design. Programming is included because it is a prerequisite; the project is included because you only really learn an idea when you try and use it. You will learn:

- **Decomposition** (breaking up big problems into simpler pieces) and **decoupling** (simplifying the relationships between those pieces)
- **Abstraction** (suppressing unimportant detail) and **specification** (how to describe an abstraction)
- **Design patterns** (solutions to common design problems)
- **Communication** (how to write code for other people, not just the machine)
- **Testing**

Our goal is to teach you how to be a software *architect*:

- not a low-level coder;
- not a debugger. We study these techniques precisely to *avoid* spending time debugging. (If you ever feel like testing or writing documentation is a waste of time, remember this – would you rather be debugging?)

4 Advice

Course strategy

- don't get behind: pace is fast!

- attend lectures: material is not all in textbook
- think in advance: don't rush to code
- de-sign, not de-bug

We can't emphasize enough the importance of starting early and thinking in advance. Of course we don't expect you to finish your problem sets the day they're handed out. But you'll save yourself a lot of time in the long run, and you'll get much better results, if you make *some* start on your work early. First, you'll have the benefit of elapsed time: you'll be mulling problems over subconsciously. Second, you'll know what additional resources you need, and you'll be able to get hold of them while it's easy and in good time. In particular, take advantage of the course staff – we're here to help! We've scheduled LA cluster hours and TA office hours with the handin times in mind, but you can expect more help if it isn't the night before the problem set is due when everyone else wants it...

Be simple:

"I gave desperate warnings against the obscurity, the complexity, and over-ambition of the new design, but my warnings went unheeded. I conclude that there are two ways of constructing a software design: One way is to make it so simple there are obviously no deficiencies and the other way is to make it so complicated that there are no obvious deficiencies."

Tony Hoare, *Turing Award Lecture*, 1980

talking about the design of Ada, but very relevant to the design of programs

How to 'Keep it simple, stupid' (KISS)

- avoid skating where ice is thin: avoid clever hacks, complex algorithms and data structures
- don't use most obscure programming language features
- be skeptical of complexity
- don't be overambitious
- Remember that it's easy to make something complicated, but hard to make something truly simple.

5 Administration and Policies

See the General Information Handout. This has detailed information about course staff, course materials, and course organization.

6 Parting Shots

Reminders:

- Complete online registration form by midnight tonight
- Get started on learning Java now!
- PS0 is due on Friday

Check this out:

http://www.170systems.com/about/our_name.html