

Lecture Notes on Software Engineering

Daniel Jackson
Fall 2002

Contents

1	<i>What 6170's About</i>	5
2	<i>Admin & Policies</i>	5
3	<i>Why Software Engineering Matters</i>	6
3.1	<i>Development failures</i>	7
3.2	<i>Accidents</i>	7
3.3	<i>Software Quality</i>	9
4	<i>Why Design Matters</i>	9
4.1	<i>The Netscape Story</i>	11
5	<i>Advice</i>	12
6	<i>Parting Shots</i>	13

Lecture 1: Introduction

1 What 6170's About

Course is actually three courses in one:

- crash course in object-oriented programming
- software design in the medium
- studio course on team construction of software

Emphasis is on design. Programming is included because it's a prerequisite; the project is included because you only really learn an idea when you try and use it.

You will learn:

- how to design software: powerful abstraction mechanisms; patterns that have been found to work well in practice; how to represent designs so you can communicate them and critique them
- how to implement in Java
- how to get it right: dependable, flexible software.

Not hacking

- how to be an architect, not just a low-level coder
- how to avoid spending time debugging

2 Admin & Policies

Course staff intros:

- Lecturers: Daniel Jackson and Rob Miller
- TAs: you'll meet in review session *next* week
- LAs: you'll meet in clusters
- Hours: see website. Lecturers don't have fixed office hours but happy to talk to students: just send email or drop by.

Materials:

- course text by Liskov; read according to schedule in general info handout
- lecture notes: usually published the day of the lecture
- 'Gang of Four' design patterns book: recommended
- 'Effective Java' by Bloch: recommended
- Java tutorial: see general information handout for details

Recommended texts are really superb; will be good references, and will help you become a good programmer faster. Special deal if you buy as package.

Course organization:

- First half of term: lectures, weekly exercises, reviews, quiz
- Second half of term: team project. More to say on this later.

A change from previous terms: no need to worry now about who will be in your team. Expect to switch TA's at half term.

Reviews:

- Weekly sessions with TAs will be used for *review* of student work
- Initially, TAs will pick fragments of your work to focus on
- Whole section will discuss in a constructive and collaborative way
- Absolutely essential part of course: opportunity to see how ideas from lecture get applied in practice

Learning Java:

- It's up to you, but we try and help
- Use Sun's Java tutorial and do exercises
- Great team of lab assistants on hand in clusters to help you

Collaboration and IP policy:

- see general info
- in short: you can discuss, but written work must be your own; includes spec, design, code, tests, explanations
- you can use public domain code
- in team project, can collaborate on everything

Quizzes:

- two in-class quizzes, focusing on lecture material

Grading:

- 70% individual work = 25% quizzes + 45% problem sets
- 30% final project, all in team get same grade
- 10% participation extra credit
- *no late work will be accepted*

3 Why Software Engineering Matters

Software's contribution to US economy (1996 figures):

- greatest trade surplus of exports
- \$24B software exported, \$4B imported, \$20B surplus
- compare: agriculture 26-14-12, aerospace 11-3-8, chemicals 26-19-7, vehicles 21-43-(22), manufactured goods 200-265-(64)

(from *Software Conspiracy*, Mark Minasi, McGraw Hill, 2000).

Role in infrastructure:

- 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.

Cost of software:

- Ratio of hardware to software procurement cost approaches zero
- Total cost of ownership: 5 times cost of hardware. Gartner group estimates cost of keeping a PC for 5 years is now \$7-14k

How good is our software?

- failed developments
- accidents
- poor quality software

3.1 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'

3.2 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
- 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; 8 died, of which 3 as result; 3/4 of surviving 20 expected to develop 'serious complications which in some cases 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
- FDA, at least, concluded that 'interpretation of beam block data by software' was a factor
- see <http://www.fda.gov/cdrh/ocd/panamaradexp.html>

Ariane-5 (June 1996)

- European Space Agency
- complete loss of unmanned rocket shortly after takeoff
- due to exception thrown in Ada code
- faulty code was not even needed after takeoff
- due to change in physical environment: undocumented assumptions violated
- see: <http://www.esa.int/htdocs/tidc/Press/Press96/ariane5rep.html>

The Ariane accident is more typical of most software disasters than the radiotherapy machine accidents. It's quite rare for bugs in the code to be the cause; usually, the problem goes back to the requirements analysis, in this case a failure to articulate and evaluate important environmental assumptions.

London Ambulance Service (1992)

- loss of calls, double dispatches from duplicate calls
- poor choice of developer: inadequate experience
- see: <http://www.cs.ucl.ac.uk/staff/A.Finkelstein/las.html>

The London Ambulance disaster was really a managerial one. The managers who produced the software were naive, and accepted a bid from an unknown company that was many times lower than bids from reputable companies. And they made the terrible mistake of trying to go online abruptly, without running the new and old systems together for a while.

In the short term, these problems will become worse because of the pervasive use of software in our civic infrastructure. 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 unpredictable 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/>

RISKS Forum

- collates reports from press of computer-related incidents
- <http://catless.ncl.ac.uk>

3.3 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.

4 Why Design Matters

“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.

Role of design and designers

- 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.

Even professional programmers delude themselves. In an experiment, 32 NASA programmers applied 3 different testing techniques to a few small programs. They were asked to assess what proportion of bugs they thought were found by each method. Their intuitions turned out to be wrong. They thought black-box testing based on specs was the most effective, but in fact code reading was more effective (even though the code was uncommented). By reading code, they found errors 50% faster!

Victor R. Basili and Richard W. Selby.

Comparing the Effectiveness of Software Testing Strategies.

IEEE Transactions on Software Engineering. Vol. SE-13, No. 12, December 1987, pp. 1278–1296.

For infrastructural software (such as air-traffic control), design is very important. Even then, many industrial managers don't realize how big an impact the kinds of ideas we teach in 6170 can have. See the article that John Chapin (a former 6170 lecturer) and I wrote that explains how we redesigned a component of CTAS, a new air-traffic control system, using ideas from 6170:

Daniel Jackson and John Chapin. *Redesigning Air-Traffic Control: An Exercise in Software Design*. IEEE Software, May/June 2000. Available at <http://sdg.lcs.mit.edu/~dnj/publications>.

4.1 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 understanding 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 quickly became the browser of choice 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 was driven by the desire to go back to small team development. Without clean and simple interfaces, it becomes impossible to divide up the work into independent groups.

Netscape set aside 2 months to re-architect the browser, but it wasn't long enough. So they planned 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. So Microsoft won the browser war, and AOL acquired Netscape.

This is not the entire story, by the way. 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.

You can read the whole story in: 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.

5 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

Can't emphasize enough importance of starting early and thinking in advance. Of course I 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 & data structures
- don't use most obscure programming language features
- be skeptical of complexity
- don't be overambitious: spot 'creeping featurism' and the 'second system effect'
- Remember that it's easy to make something complicated, but hard to make something truly simple.

Optimization rule

- Don't do it
- For experts only: Don't do it yet

(from Michael Jackson, *Principles of Program Design*, Academic Press, 1975).

6 Parting Shots

Reminders:

- Tomorrow is a *lecture* not a review session
- Complete online registration form by midnight tonight
- Get started on learning Java now!
- Exercise 1 is due next Tuesday

Check this out:

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