# 6170 Lecture Notes
Daniel Jackson, Fall 2k

# Lecture 1: Introduction

## 1.1 Course overview

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 at this stage, you don't really understand designs until you implement them.

You will learn:
- how to design software using some powerful abstraction mechanisms and a collection of patterns that have been found to work well in practice;
- how to get it right, by construction and by modular reasoning;
- how to articulate your design ideas and critique other people's designs;

and on the way:
- how to to think about a problem
- how to code in Java
- how to work in a team

Materials
- new textbook by Liskov
- lecture notes
- other recommended stuff on website (eg, Sun's Java documentation)

Prerequisites
- 6001, or some experience programming in a high-level language

### 1.1.1 Course organization and policy

Course in two halves. First half of term is lectures and weekly assignments done individually; second half is team project. Problem sets for first half build up to implementation of MapQuick, a local version of MapQuest using the US Census Bureau Database. Team project is Gizmoball, with some new and exciting features.

What we expect from you:
- attend lectures and recitations;
- complete readings in advance of lectures;
- do problem sets weekly for first half of term;
- do closed-book quiz at half term;

- attend project reviews;
- complete design and implementation of project.

What you can expect from us:
- lectures that explain new ideas and show how to apply them;
- recitations that offer practice and critique;
- problem sets that increase your skill with minimal grunt work;
- timely grading of your work;
- openness of lecturers to chat (drop by, and send email);
- availability of TA's during office hours;
  availability of lab assistants online;

Collaboration and IP policy:
- you may talk about course material with your colleagues;
- you may not share insights into weekly assignments;
- you may use any code we provide, and any code in the standard Java library;
- you may copy code and algorithms from textbooks or from general online sources;
- you may not copy each other's code, or use code written in 6170 by previous students;
- in the team project, you may share everything but should all contribute to all aspects.

Grading
- 75% on individual work: 45% problem sets, 25% quiz, 5% recitation participation
- 25% on team project
- we reserve the right to normalize across sections

Late policy
- no credit for late work
- but one slack weekend: can hand in Monday at noon instead of Friday at noon
- can't use slack on final project

Completion credit
- for problem sets 1 to 5, some part of the grade is for completing the implementation
- that means documenting, testing too
- can get this credit later if you weren't awarded it the first time
- you'll need work from earlier problem sets in later ones

Programming diagnostic
- to help us gauge your background
- due on Friday (September 8)
- not graded, but you cannot take the course unless you complete it

Signup sheet
- due at end of class today

### 1.1.2 Why does software engineering matter?

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 Consipracy*

Role in infrastructure
- not just the Internet
- transportation, energy, medicine, finance

How good is our software?
- failed developments
- accidents
- poor quality software

## 1.2 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

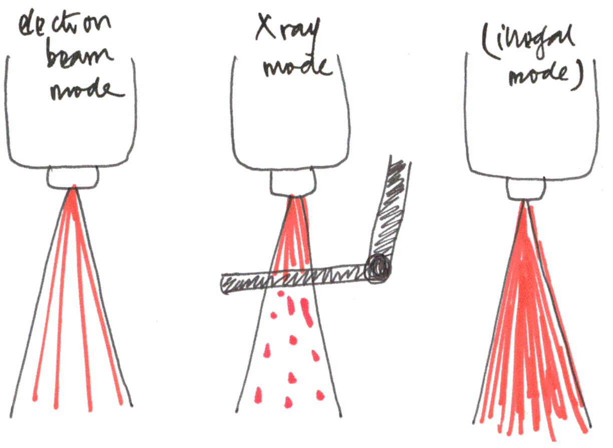
Bureau of Labor Statistics (1997)
- for every 6 new systems put into operation, 2 cancelled
- probability of cancellation is about 50% for biggest systems
- average project overshoots schedule by 50%
- 3/4 systems are regarded as 'operating failures'

## 1.3 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*
*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



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://archive.eiffel.com/doc/manuals/technology/contract/ariane/page.html

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/ (Resources section)

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 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.nitrd.gov/pitac/report/

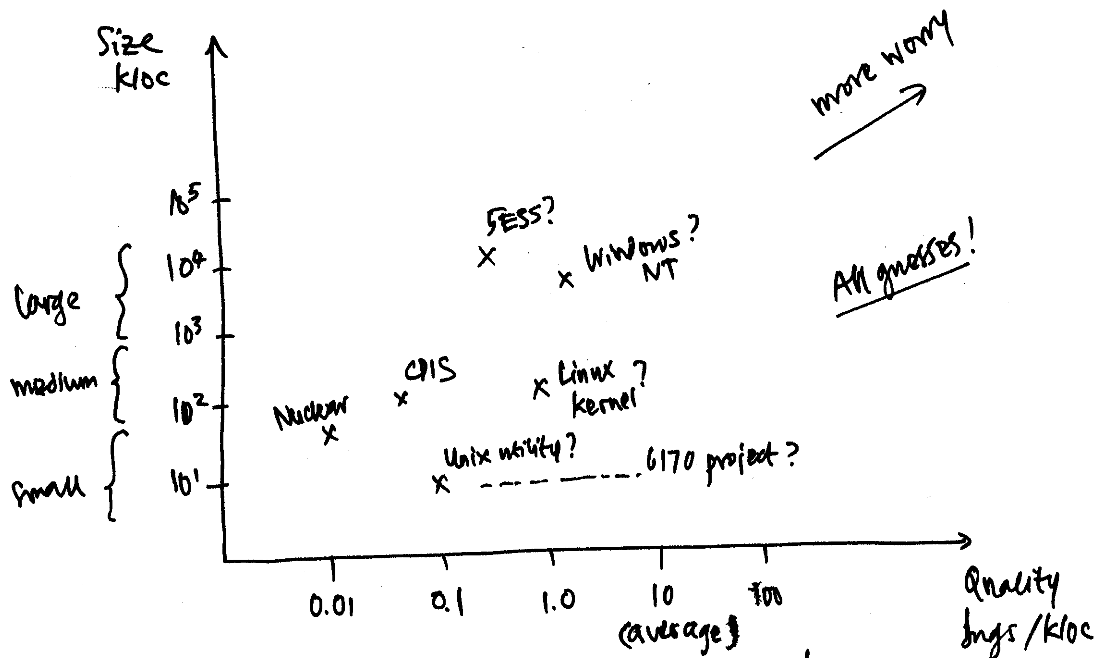
RISKS Forum
- collates reports from press of computer-related incidents

- http://catless.ncl.ac.uk

## 1.4 Software Quality

One measure: bugs/kloc
- measured after delivery
- industry average is about 10
- high quality: 0.1 or less

Some rough guesses:



Praxis CDIS system (1993)
- UK air-traffic control system for terminal area
- used formal methods: precise specification
- no increase in net cost
- much lower bug rate: about 0.75 defects/kloc
- even offered warranty to client!

Sample contracts:
- Cosmotronic Software Unlimited Inc. does not warrant that the functions contained in the program will meet your requirements or that the operation of the program will be uninterrupted or error-free. However, Cosmotronic Software Unlimited Inc. warrants the diskette(s) on which the program is furnished to be of black color and square shape under normal use for a period of ninety (90) days from the date of purchase.

- We don't claim Interactive EasyFlow is good for anything – if you think it is, great, but it's up to you to decide. If Interactive EasyFlow doesn't work: tough. If you lose a million because Interactive EasyFlow messes up, it's you that's out of the million, not us. If you don't like this disclaimer: tough. We reserve the right to do the absolute minimum provided by law, up to and including nothing. This is basically the same disclaimer that comes with all software packages, but ours is in plain English and theirs is in legalese.

*ACM Software Engineering Notes, Vol. 12, No. 3, 1987.*

## 1.5 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!
- 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

It's a funny thing that computer science students areoften 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 (another 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 17, no. 3 (May/June 2000).*

## 1.6 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.

## 1.7 Advice

Course strategy
- don’t get behind: first week especially is very fast!
- attend lectures: material is not all in textbook
- do the readings on time

Life strategy
- think in advance: don’t rush to code
- design is more fun than debugging!
- focus on ideas, not embodiments
  - don’t be blinded by technology
  - you should master Java, but realize that it will become obsolete

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*

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
- don’t optimize until proven necessary
- 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.

## 1.8 Closing Admin

What you must do:
- Hand in sign-up sheet before you leave.
- Complete and hand in PS0 (programming diagnostic) by Friday.

Recitation: tomorrow, we'll send section assignments to you by email.

Help getting going with Java
- Electronic classrooms open Sunday and Tuesday, with TA's on hand to help