Inf2C: Software Engineering

Paul Jackson

School of Informatics
University of Edinburgh

About this course

This course has two main jobs:

- give you an overview of what software engineering is
- take you beyond programming to engineering software

This is a tall order for one 10pt course!

Why do this course?

Because software engineering is fascinating

blend of human and technical challenges

fast-moving

important

It could help you get a job!

Teaching style

Lectures provide an overview (not self-contained notes)

- Self-study of course topics and Java essential
- Pointers provided in slides and on website

Tutorials with question sheets (Weeks 4,6,8,10)

Coursework: a project in three parts (40%)

Labs with demonstrators

Online discussion forum

Email

Exam: short-answer questions (60%)

Books

No book is essential.

The following are worth considering:

Sommerville, *Software Engineering*

- Large, classic. Comprehensive on SE, but limited on UML and Java.

Stevens with Pooley, *Using UML*

- Covers basic SE, does UML thoroughly, no Java.

Why is software engineering still hard?

Easy (or at least routine) projects

small systems (up to c. 100k LOC),

- without hard timescales or budgets,

- without requirement for very high reliability,

- without complex interfaces or legacy requirements [...]

Hard projects

everything else. Projects with *all* the above challenges, and more.

Statistics

The Standish Chaos reports on medium-large organisations classify software development projects:

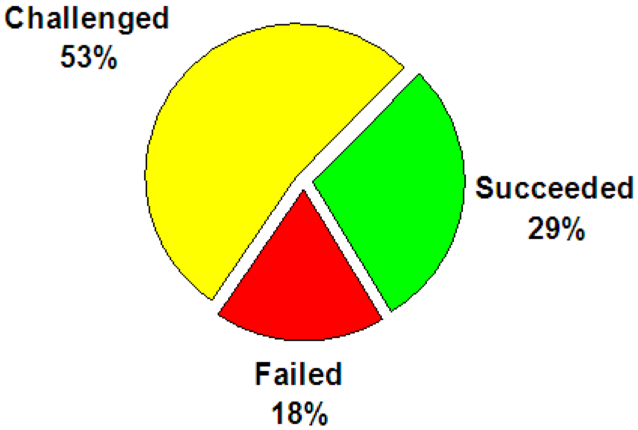
- Succeeded
  1994: 16% ... 2004: 29% ...2009: 32%)
- Challenged (i.e., delivered something but maybe reduced scope, late, over budget)
  no real trend, around 50%
- Failed (i.e., cancelled without delivering anything)
  1994: 31% ... 2004: 18% ...2009: 24%)

Methodology not perfect, but statistics are indicative.

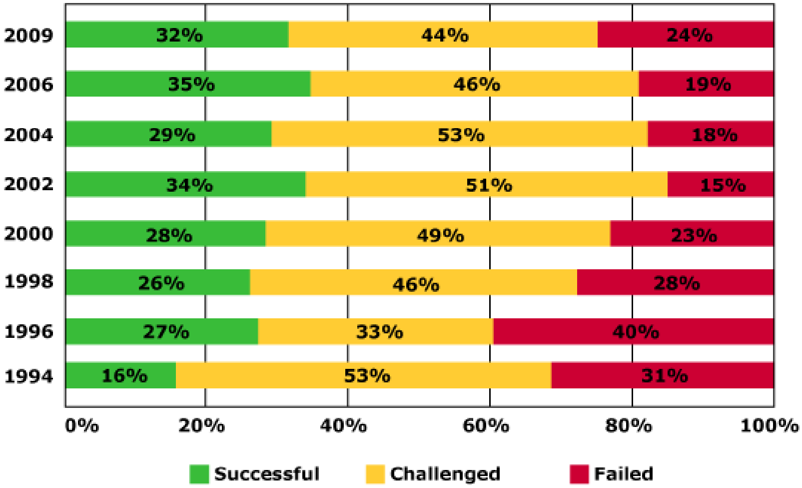
CHAOS 2004

Survey Results

Resolution of Projects



Standish Chaos trends to 2009



The fundamental tension

control $\leftrightarrow$ flexibility

Historically a natural tendency to tackle problems such as

- uncertain requirements
- overruns of time or budget
- keeping stakeholders in sync

with ever greater control. But greater control means

- more planning,
- more documentation,
- tighter management

More *ceremony*.

1990's/2000's backlash: agile methods, e.g. Extreme Programming, with slogans like "Embrace Change".

Deliberately *low ceremony*.

In this course we try to give you a flavour of both approaches.

Software engineering activities

Syllabus list:

- *requirements capture*
- *design*
- *construction*
- *testing, debugging and maintenance*
- *software process management*

software development process: How these activities are ordered and related

Requirements capture

Identifying what the software *must do* (not *how*). Recorded using a mixture of *structured text* and *use case diagrams*.

Interesting issues:

- Multiple stakeholders often with different requirements – how to resolve conflicts?
- Prioritisation. Which requirements should be met in which release?
- Maintenance: managing evolving requirements.

Techniques: use e.g., case analysis, viewpoint analysis, rapid prototyping.

Design

Requirements: what the software must do.

Design: how should it do that?

Higher level than code. Often recorded using a modelling language e.g. UML (*Unified Modeling Language*).

Multiple levels of design:

- architectural design
- high-level design
- detailed design

Interesting issues: understandability ("elegance"); robustness to requirement change; security; efficiency; division of responsibility ("buildability").

Techniques: e.g., introspection, reviews of various kinds, design patterns, Class-Responsibility-Collaboration (CRC) cards...

Construction/implementation

More general than "coding", includes:

- detailed design (the level that doesn't get written down)
- coding
- unit testing
- "hygiene" tasks like configuration management
- developer-targeted documentation

Interesting issues: scale: managing large amounts of detail, esp. code. Need systems that work when it's not possible for one person to know everything.

Techniques: Lots of software tools...

Testing and debugging

Testing happens at multiple levels, from unit tests written before coding by developer, to customer acceptance testing.

Debugging covers everything from “which line of code causes that crash?” to “why can’t users work out how to do that?”.

Interesting issues: containing cost – how to test and debug efficiently; software tools to support testing and debuggin

Techniques: software tools e.g. JUnit, Selenium, IDE debugger.

Maintenance

Any post-(major)-release change.

1. corrective maintenance (bugfixing!)
2. perfective maintenance (enhancing existing functionality)
3. adaptive maintenance (coping with a changing world)
4. preventative maintenance (improving maintainability)

Traditionally an after-thought - mistakenly!

In the "total cost of ownership" (TCO) of software system, maintenance costs often dwarf development costs.

Interesting issues: retaining flexibility; when to refactor/rearchitect/retire/replace system

Techniques: e.g., refactoring

Software process management

Meta-level activity. How can a group of people carry out all these activities so as to produce software that customers are happy to pay for?

How should the activities be structured? E.g. all requirements analysis first, or just enough to do the first bit of design?

Interesting issues: balancing flexibility against controllability, producing just enough paper; enabling continual improvement of process.

Techniques: e.g., reviews, various kinds of certification, Capability Maturity Model.

Software engineering discipline

What is a software engineer, as distinct from a programmer?

E.g.

- someone who isn't going to be surprised when the customer turns round and wants something else.
- Someone who's thought about/been educated in the wider software engineering issues such as ethics.

Software engineering is a (relatively) young discipline. Is it "engineering"?

What does a software engineer need to know? What must they be able to do? IEEE SWEBOK; SE2004 curriculum; etc.

Should software engineers be chartered? Should they be legally required to be?

Ethics – doing good

As software becomes ever more pervasive, software engineers are increasingly called on to make ethical decisions.

As responsible (chartered?) professionals, they should act in the public good, and avoid acting unfairly, harmfully, and illegally.

The ACM and IEEE have written a Software Engineering Code of Ethics and Professional Practice:

`http://www.acm.org/about/se-code`

> *Consider writing software for car engine control that must minimise emissions and maximise fuel economy.*
>
> *What if doing both at the same time is costly, but emissions tests can be automatically detected?*
>
> *A bad decision on this is likely to cost VW*
>
> *$18bn*

Learning outcomes

- Explain how to apply commonly agreed ethical principles to a software engineering situation.
- Motivate and describe the activities in the software engineering process.
- Construct use cases for the system requirements.
- Explain and construct UML class diagrams and sequence diagrams.
- Understand and construct a software system using Java. Assess the software system using testing and other appropriate tools.
- Evaluate aspects of human usability of an application program or web site.
- Compare different approaches to software licensing.

Reading

Aim: deepen your understanding of what software engineering is and why the term was invented and is still used, and why problems still exist.

Compulsory: Read the ACM/IEEE Ethics code `http://www.acm.org/about/se-code` and think about cases where the principles might conflict.

Suggested: google software engineering ethics.

Suggested: browse the proceedings of the NATO conferences on Software Engineering (see web page).

Suggested: Sommerville Chapter 1 and/or Stevens Chapter 1.

Suggested: google Chaos Standish reports, find e.g. `http://www.infoq.com/articles/Interview-Johnson-Standish-CHAOS`