IRM: Software Engineering

why might it matter to you?

Perdita Stevens

School of Informatics
University of Edinburgh

Plan

- The problems of software engineering
- Research in software engineering
- Software engineering in research

- biased towards my interests, may be hijacked

Do feel free to interrupt.

The problems of software engineering

In a nutshell:

To build ever-more complicated systems faster, better, cheaper.

Practising software engineers and software engineering researchers are working in different ways on the same problems.

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 [...]

Hard projects

everything else. Many projects have *all* the above challenges, and then some.

Statistics

The Standish Chaos reports have since 1994 collated information from medium to large government and commercial organisations, classifying their software development projects into

- Succeeded
- Challenged (i.e., delivered something but maybe reduced scope, late, over budget)
- Failed (i.e., cancelled without delivering anything)

Methodology has been questioned, and NB “hard” projects overrepresented. Still...

Statistics

The Standish Chaos reports have since 1994 collated information from medium to large government and commercial organisations, classifying their software development projects into

- 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 has been questioned, and NB “hard” projects overrepresented. Still...

The fundamental tension

control $\leftrightarrow$ flexibility

Natural tendency to tackle problems of e.g. uncertain requirements, overruns of time or budget by ever more control or *ceremony*: more planning, more documentation, tighter management...

Recent backlash: agile methods, e.g. Extreme Programming, with slogans like "Embrace Change". Deliberately low ceremony.

A few points of consensus

There are many competing software development methodologies with their own strengths and weaknesses.

However, we now understand that:

- Component-based design is the new OO
- Architecture is vital to reuse and more
- Quality doesn't just happen
- A defined software process is a double-edged sword
- Iteration will happen anyway, so best to plan it

SE research methodologies

Software engineering is as broad as computer science – and it's as provocative to call it "engineering" as it is to call CS "science". In different parts of the field, basic methodologies are:

- Mathematics
- Experiment
- Historico-sociological argument, e.g., supported by ethnography. (Often mixed with experiment: controlled experiments are very problematic in SE.)
- "Suck it and see" or *advocacy*

(By a "research methodology" I mean a method by which you convince someone that your work contributes.)

Highly recommended further reading

What Makes Good Software Engineering Research

http://www.cs.cmu.edu/∼Compose/ftp/shaw-fin-etaps.pdf

Writing Good Software Engineering Research Papers

http://www.cs.cmu.edu/∼Compose/shaw-icse03.pdf

Software engineering in research

In informatics (and elsewhere), doing research often involves building software.

e.g.

- to demonstrate a new algorithm
- to compare several techniques for solving the same problem
- to play with a theory or idea
- to control hardware that runs physical experiments
- to analyse results.

Software engineering in academia

Surely, it's a bug that there should be any difference between software engineering in academia and software engineering in industry?

As responsible academics, we should take a best-practice software development methodology off the shelf and use that?

NO!

Why academia is different

1. You're working in very small teams, often teams of 1.
2. You don't have a customer (really).
3. You're not a full time software developer.
4. Your risks are mainly technical.

There are other differences that often, but not always, apply. For example, what is the probability that someone other than you will have to maintain your system?

Different problems, different solutions

Major problems of industry:

- Customers' requirements are not initially known by anybody, and change rapidly compared to system development times
- It is essential but often very hard to understand the domain
- Software is typically developed by teams of dozens or hundreds of developers, maybe spread round the world, maybe over many years
- There is a high turnover of software developers
- Documentation needed to avoid argument later.

So it’s OK to hack?

Well, plenty of academics do it and still get the publications...

But there are reasons not to.

Once upon a time... I came to Edinburgh to take over the Edinburgh Concurrency Workbench.

Based on that experience, I want to talk about some things that are worth doing even in academia.

Testing

You will make mistakes. You will not always notice them at once.

Unless you really enjoy debugging, consider:

- automated system testing: design it in
- framework-supported unit testing: consider something like JUnit
- if you're going to let anyone else near your system, usability testing

(You will also want to be able to demonstrate how great your system is: that's a different and easier problem.)

Design

I think the most relevant methodology here (and elsewhere) is Extreme Programming (XP).

`http://www.xprogramming.com`

Creative tension between simplicity and rigour:

1. Do the simplest thing that could possibly work
   You Ain't Gonna Need It
2. Keep the system well-designed all the time
   Refactor mercilessly.

User interface design

This matters a lot more than you'd think!

Make sure you have a pretty multicoloured GUI for conference demonstrations and for screenshots in your papers...

... and make sure it's actually usable.

Designing a basically usable UI isn't rocket science, just needs to be thought about – it's embarrassingly easy to make basic errors.

Read a good book – I like *User Interface Design* by Harold Thimbleby – and watch people learning to use your system, before you let people download it remotely.

Project management

A small problem when it's just you! But

Things take longer than you think.

(Humans are systematically optimistic when they make estimates of how long anything will take – not just software! They assume there will be no interruptions or unforeseen problems.)

Danger of spending weeks/months on building something that, when it works, doesn't achieve quite what you intended.

Consider splitting the task into tiny fragments, beginning with the most valuable. E.g., have a running system that passes your tests every day.

Reuse

- of components and infrastructure, e.g. GUI toolkits (e.g. gtk), graph-drawing packages, diagramming packages (e.g. GEF), lexers and parsers,...
- of domain specific software, e.g. [bits of] other people's tools.

Consider this early, e.g., when you decide on a programming language.

But don't underestimate the effort required.

Release management(?)

You will at some point have to choose whether to

- release your software commercially and make your fortune on earth; consider asking our commercialisation people for help;
- make your software open source and take your chances on rewards in heaven.

or neither, or something in between.

At this point if not before you will need documentation.

Running an open source project

Good way to get known and make progress, and have no time to have a life. Consider:

- what's in it for you
- configuration management
- documentation
- mailing lists
- defect tracking
- licensing

Version control

Basic ingredient of configuration management: important even on tiny projects and to individuals.

“Check in” changes with comments on what changed.

Tools such as SCCS, RCS, CVS, ... allow you to revisit previous versions, check what changes have been made, view change history, etc.

Extremely useful for backtracking when something goes wrong; means you can try something out knowing you can easily go back.

E.g. Emacs vc-mode makes this very easy: see VC submenu of Tools menu.

Version control what?

Personally I version control just about everything that might change –

source code, tests, test results, papers, book chapters, CV, frames, .emacs, ...

and it has saved me considerable hassle many times.

Low overhead, high value

Further reading

There is, of course, a vast and easily accessible literature on this topic on the web. My home page has some starting points.

Much of what I've said is said at greater length in a paper

A Verification Tool Developer's Vade Mecum

`http://homepages.inf.ed.ac.uk/perdita/vademecum.ps`

I strongly recommend browsing

`http://www.xprogramming.com`

for more info on XP.