LECTURE 1: INTRODUCTION

Software Engineering

Mike Wooldridge

- The problem is *complexity*.
- Many sources of complexity, but *size* is key:
  - UNIX contains 4 milliion lines of code
  - Windows 2000 contains 10^8 lines of code

  "If steel girders could be infinitely long, and didn't bend no matter what you did, then buildings could be as large and complex and computer systems."
  (Brian Reid)
- Software engineering is about managing this complexity.

1 Why Software Engineering?

- Software development is *hard*.
- Important to distinguish "easy" systems (one developer, one user, experimental use only) from "hard" systems (multiple developers, multiple users, products).
- Experience with "easy" systems is misleading.
  One-person techniques do not scale up.
- Analogy with bridge building:
  — over a stream = easy, one person job
  — over River Severn...
  The techniques don't scale.

2 Goals of Software Development

- Satisfy users requirements.
- High reliability.
- Low maintenance costs.
- Delivery on time.
- Low production costs.
- High performance.
- Ease of reuse.

Note importance of *tradeoffs*.

2.1 Satisfying User Requirements

- Many programs simply don't do what end users want.
- Typical percentages for large-scale commissioned systems:

  45% delivered but not used
  27% paid for but not delivered
  17% abandoned
  6% used after changes
  5% used as delivered

- Users find it hard to articulate what they want.
- Developers find it hard to understand what users say!

2.2 High Reliability

- Mistakes in programs are generically known as *bugs*.
- A crucial lesson:

  You can prove that bugs are there; you can't prove that they aren't.

- Bugs can be expensive, in terms of...
  - *human lives*:
    in safety critical systems, e.g., nuclear reactor control, fly-by-wire aircraft
  - *money*:
    software bug in failed Ariane 5 launch cost US$500 million
  - *poor customer relations*:
    Microsoft problems with original Windows release caused the company huge problems.

2.3 Low Maintenance Costs

- Maintenance is what is done to software after it starts being used.
- Maintenance may be:
  - *corrective* — fixing bugs (21%);
  - *adaptive* — altering software to fit changing software (25%);
  - *perfective* — to meet new requirements (50%);
  - *preventative* — to reduce further maintenance (4%).
- Maintenance is expensive — much software is "finely balanced", with apparently small changes having a major impact.

  (Imagine being a car mechanic, and having to figure out from scratch how the engine works every time you start work on a new car.)
- Maintenance typically accounts for 65% of overall project costs.

2.4 Delivery on Time

- Software projects are notorious for overrunning.
- It is extraordinarily hard to reliably predict how much effort a software project will require, and when it will be completed.
- *The relationship between person months devoted and development time is almost never linear*:
  - adding person months of effort to a project frequently has no effect;
  - adding person months of effort often makes the project slower.

2.5 Production Costs

- Software production an enormous industry:
  - in 1985, US$70 billion spent on development in 1985
  - OS/360 cost US$200 million
  - estimated expenditure for year 2000: US$770 billion
  - 12% growth per annum!
- Software developer skills in poor supply ⇒ expensive.

2.6 High Performance

- Systems that are specially tailored to work quickly are *optimised*.
- Optimization can improve throughput (speed) and memory usage, but:
  - resource intensive;
  - results in systems tailored to a specific environment;
  - less comprehensible to developers;
  - less easy to change;
  - less easy to *port* to other environments.

2.7 Ease of Reuse

- Goal of software reuse: use same software in different systems and software environments:
  - reduce development costs;
  - improve reliability.
- Most software has two parts:
  - environment independent part — can be moved between environments quite easily (usually includes the "logic" of the system);
  - environment dependent part — cannot be moved easily (includes e.g., GUI, hardware controllers, ...)

  A clear separation between the 2 is crucial.
- Requires *designing for reuse*.

3 The Software Process

- The waterfall model of the software lifecycle:

- *Requirements analysis and definition.*
  - The system's services, constraints, and goals are established.
  - Requirements analysis means long consultations with the end-user to establish exactly what they want.
  - Requirements definition means stating what the user wants, in terms that are understandable by both end-users and system developers; relatively informal.
  - Requirements specification is a more formal statement which sets out proposed system services in detail. This document may act as contract between system procurer and developer.
  - Software specification is an abstract description of system structure & operation, intended to serve as the basis for the design stage.
  - *Deliverables*: requirements definition document, requirements specification document & software specification document.

- *System and software design.*
  - Requirements are divided into those relating to hardware and those relating to software.
  - Software design then means representing the functions of each unit in a way that may be transformed into code.
  - *Deliverables*: unit and system designs.

- *Implementation and unit testing.*
  - Unit designs are transformed into programs.
  - Individual units are then tested, to ensure that they satisfy their specification.
  - *Deliverables*: implemented and tested unit programs.

- *System testing.*
  - Individual programs & units are integrated (gradually!) and tested to ensure that system requirements have been met.
  - The system is then installed.
  - The system is then maintained.
  - *Deliverables*: implemented, tested system.

4 Other Development Models

- *Prototyping*:
  - develop "quick and dirty" system quickly;
  - expose to user comment;
  - refine;

  until adequate system developed.

  Particularly suitable where:
  - detailed requirements not possible;
  - powerful development tools (e.g., GUI) available.
- *Formal transformation*:
  - inolves use of mathematical methods for specification, development, verification;
  - despite several decades of effort, not usable without special skills;
  - used in certain applications (e.g., verification of DS1 controller).

5 Challenges for Software Engineering

- *The legacy challenge.*
  - Hardware evolves *much* faster than software.
  - Most software systems in use today were developed many years ago.
  - They are technologically obsolete (cf COBOL Y2K problem) but perform business-critical tasks.
  - Frequently, nobody understands how the software works.
  - Modifications over the years have meant that the software logic has become corrupted and confused.
  - Original developers have moved on.
  - How to manage, maintain, replace, integrate this software?

- *The heterogeneity challenge.*
  - Isolated software systems — once the norm — are now the exception.
  - Most commercial systems are now networked.
  - Implies that software systems must cleanly integrate with other *different* software systems, built by different organisations & teams using different hardware and software platforms.

- *The delivery challenge.*
  - Software projects are notorious for being overdue and over budget.
  - The delivery challenge is about consistently being able to deliver systems on budget and on schedule.
  - As the complexity of systems that we develop increases, this challenge becomes harder.

6 Professional Issues

- Bridge builders and other types of "conventional" engineer are acutely aware of their professional responsibilities.
- Software engineers have professional responsibilities as well:
  - *Confidentiality*.
    Engineers should respect the confidentiality of their employers and clients.
  - *Competence*.
    Engineers should not misrepresent their level of competence.
  - *Intellectual property rights (IPR)*.
    These relate to who owns *ideas*.
    Typically, your employer does! You need to understand laws governing these, and how they relate to the work you are doing.

  - *Computer misuse*.
    Software engineers should not misuse their skills to misuse other people's computers.