Trial clicker question :-)

1. I've got my clicker
2. I've got my clicker and I've turned it on
3. Oh yes, turn it on
4. I've got it and turned it on but it still doesn't seem to work
5. I forgot to come and collect my clicker

Inf2C: Software Engineering

Perdita Stevens

School of Informatics
University of Edinburgh

About Inf2C

Inf2C is a 20-credit course covering software engineering and systems.

The two threads are taught more or less independently – Perdita Stevens doing software engineering, and Stratis Viglas with Marcelo Cintra doing systems.

We will alternate threads week by week (roughly)

Each thread has independent practicals, but there is only one exam.

Notes, practicals etc. will appear on the course web site.

About this course

This is the Software Engineering half of Inf2C.

- 15 lectures, introduction to software engineering.
- Java programming in the large, using IDE (Eclipse).
- Basic modelling using the Unified Modeling Language (UML).
- Basic SE techniques, and when and why to use them.

Learning outcomes

- Motivate and describe the activities in the software engineering process.
- Construct use cases for an application scenario.
- Explain and construct UML class diagrams and sequence diagrams.
- Build, document and maintain large Java programs using a modern IDE, rapid development methods, and configuration management tools.
- Explain how a software system and its construction may be assessed using testing, metrics, and verification techniques.
- Evaluate aspects of human usability of an application program or web site.
- Judge the security risks in software construction and show how to avoid or reduce them.
- Compare different approaches to software licensing.

Why do this course?

Because software engineering is fascinating :-)

– blend of human and technical challenges; fast-moving; important

Because for many of you, it's probably the most job-relevant course you'll do in your Informatics degree. Worth learning it well enough to remember long-term!

Teaching style

Lectures as guidance and overview (not self-contained notes).

You will need to do considerable study on your own, mostly from the web. As well as the explicit course material, you will need to extend and improve your Java skills.

There will be some URLs which you must visit and read. You should not need to print them, but you should make your own notes of key points. The lectures will provide questions to help test your understanding of the material, but you must know more than the answers to the questions!

Practical coursework: a single exercise, in two parts

Exam: multiple choice and short answer (covering all of Inf2C).

Support: Tutorials. Newsgroup `eduni.inf.course.inf2c`. Email only if your query is personal or confidential.

Clickers

During the SE lectures of Inf2C, we'll use "clickers". The aim is to:

- give us some information about what's clear and what's not
- let you practise multiple choice questions like those in the exam
- check that something's being absorbed from the compulsory reading
- engage brains during lectures!

Not individual assessment (in fact I will not know who's got which clicker).

This will only work if you remember to bring them!

Books

No book is essential.

The following are worth considering:

Somerville, *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.

You should already have Deitel & Deitel's Java book.

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% ...?)
- 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% ...?)

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

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.

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.

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

Software engineering activities

Syllabus lists:

> *requirements capture; design; construction; testing, debugging and maintenance; software process management.*

How these activities are ordered and related depends on the software development process used.

Let's briefly consider each in turn.

Requirements capture

Identifying what the software must do (not how). Recorded using 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 changing requirements.

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

Design

Given that we know what the software must do, how should it do it?

Higher level than code. Recorded using a modelling language e.g. UML.

Multiple levels of design:

- architectural design
- high-level design
- detailed design

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

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

Construction

Term intended to be a bit more general than "coding", to imply that it includes:

- detailed design (typically, 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: use of various tools...

Testing and debugging

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

Debugging (here) 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; avoiding own-stuff bias – how to see problems in things you did.

Techniques: lots... here, tools e.g. JUnit, use of debugger.

Maintenance

Term used for 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 viewed as unglamorous but vital. In the "total cost of ownership" (TCO) of software system, maintenance costs often dwarf development costs.

Interesting issues: retaining flexibility (avoiding architectural degradation); when to refactor/rearchitect/retire/replace system

Techniques: refactoring,...

Software process management

Meta-level. 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: 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.

Software engineering is a (relatively) young and controversial discipline.
'Engineering': snake-oil, or accurate description?

What does a software engineer 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?

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.

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

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

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

Compulsory: Read the assignments (on web page) – will discuss next time