Lecture 23: Course Summary

Course Goals

Summary of what we covered

Feedback questions for you

Sample Exam Question

Course Rationale

The goals for this course were:

- introduce the main ideas of software engineering
- offer practical experience of developing large software systems
  - especially evaluating and modifying software developed by others
- raise awareness of the need for a disciplined approach
- build on your existing experience with programming

Approach

- Typical Software Engineering courses generally:
  - introduce the issues of software engineering at a high (theoretical) level
  - follow a waterfall lifecycle through the main phases
  - introduce one analysis and design method in detail with a team project
- Problems with these courses
  - students do not get sufficient experience of the difficulties of large scale software development and maintenance
  - students learn how to use the techniques, but don't gain an appreciation of why they are useful
- Hence, the trading game…

Course Outline

Introductory stuff

- Case Studies
- Software Lifecycles
- Project Management, Risk Management

Program Design

- Decomposition and Abstraction
- Procedural Abstraction
- Data Abstraction
- Software qualities; modularity
- Design Representations

Verification & Validation

- Testing
- Reviews & Fagan Inspections
- Formal verification
- Debugging and exception handling

Software in the large

- Requirements Engineering
  - Structured Analysis
  - Object Oriented Analysis
  - Formal Analysis
- Specifications
- Software Architectures
- Software Maintenance
  - evolution
  - reengineering
  - reuse
- Software Process Modeling
  - process improvement
  - capability maturity
- Software Measurement

Conclusions

Key lessons

- software development is far more than just writing programs
- communication is more important than coding
- testing and inspection are vital for quality assurance
- software engineers need to reflect upon their own development processes and seek to improve them continuously

Key skills

- judging software quality
- reading/modifying other people's code
- working with vague or incomplete specifications
- working to tight (impossible!) deadlines
- working with changing requirements/constraints
- communicating about technical work
- negotiating contracts to buy (and sell) software
- working in teams
- learning from mistakes (and learning to reflect on your experiences)
- deciding how much and what types of documentation are helpful
- deciding what is important (because perfect software is impossible)

Feedback Questions

Did the course meet your expectations?

How useful do you think the course was to you?

What do you feel you have learned?

What did you *not* learn, that you had hoped to?

What was the best part of the course?

What was the worst part of the course?

How might the course be improved in the future?

Sample Exam Question

1 a) Why is random testing insufficient even for relatively small programs?
[2 marks]

b) *Unit testing* is the process of testing a single program unit (e.g. a procedure) in isolation from the rest of the program. How would you go about choosing test cases for unit testing? [4 marks]

c) *Integration testing* can be tackled top-down or bottom-up. Describe each of these strategies. Why is integration testing harder than unit testing?
[4 marks]

d) Explain the purpose of each of the following. What types of error is each likely to find?

i) Endurance testing ii) Recoverability testing iii) Regression testing
[6 marks]

e) The company you work for develops internet applications. To reduce time to market, the company is considering dispensing altogether with integration testing. Instead, the company plans to rely on Beta testing, in which free trial versions of new software will be sent to existing, trusted customers to try out, with the agreement that they will report any problems they encounter. What are the advantages and disadvantages of this approach?
[4 marks]

How we grade it...

a) Why is random testing insufficient even for relatively small programs? [2 marks]

2 marks for a detailed explanation, 1 mark for a partial answer. Several possible reasons:

- Each decision point in the code represents a branch. As the number of decision points grows, the number of possible paths through the code grows exponentially. Random choices of test data is unlikely to cover all paths.
- Most of the interesting errors in software occur for particular data points, e.g. on the boundaries between different input ranges. Choosing test data randomly is unlikely to hit the boundary conditions.
- To properly test software, you need to define its operational profile (i.e. how frequently it is likely to see each type of input/behaviour). Random selection of test cases is unlikely to match the operational profile.

b) *Unit testing* is the process of testing a single program unit (e.g. a procedure) in isolation from the rest of the program. How would you go about choosing test cases for unit testing? [4 marks]

4 marks for four different ways of choosing test cases OR two different ways of choosing test cases together with a good explanation of why each approach is good. Can give one mark for talking about the difference between black and white box testing, but needs more specific ways of choosing test cases to get more marks:

- Boundary conditions
- Normal behaviours
- Off-nominal cases (inputs that the program is not supposed to be able to handle)
- Parameters in the wrong order
- Different 'paths' through the specification
- Test each branch
- Test each conditional statement

How we grade it...

c) *Integration testing* can be tackled top-down or bottom-up. Describe each of these strategies. Why is integration testing harder than unit testing? [4 marks]

1 mark for describing each strategy. 1 extra mark for making the difference clear, or for describing advantages of each. 1 mark for saying why integration testing is harder.

Top-down: test the top level ('main') procedure first, with stubs for each procedure it calls. Stubs should check whether parameters passed downs are okay, and return some test data. Then integrate the next level of procedures and test again, repeat until you've integrated the bottom level procedures

Bottom-up: first test those procedures that don't call any others. Then integrate & test the procedures that call the ones you've tested, repeat until you reach the top level (main) procedure.

Integration testing is harder because it is impossible to ensure every path through the integrated system is tested, it's much harder to track down errors, and interface errors that show up in integration testing tend to be more subtle.

How we grade it...

d) Explain the purpose of each of the following. What types of error is each likely to find?

i) Endurance testing

ii) Recoverability testing

iii) Regression testing [6 marks]

1 mark for explaining each of three types, 1 mark for describing the types of error each will find.

Endurance testing means leaving the system running for long periods of time. It will catch errors that show up only after a long run, e.g. memory leaks.

Recoverability testing tests how well the software can recover from bad data, from hardware failure, from failure of systems it interacts with, from failure of components within the software. Type of error found are where data (e.g. file system) gets corrupted and cannot be recovered, program can't be re-started after a crash, etc.

Regression testing means running all the tests again (even those that already passed) each time the software is modified. This catches errors that are introduced as the result of fixing other errors.

How we grade it...

e) The company you work for develops internet applications. To reduce time to market, the company is considering dispensing altogether with integration testing. Instead, the company plans to rely on Beta testing, in which free trial versions of new software will be sent to existing, trusted customers to try out, with the agreement that they will report any problems they encounter. What are the advantages and disadvantages of this approach? [4 marks]

2 marks for good advantages. E.g:

- Cheaper
- May be able to get the software to market quicker
- Generates early interest in the software, lets users know its on the way.
- Real users are more likely to try out typical patterns of usage
- Real users are more likely to try doing dumb things to the software
- Real users will try out the software on all sorts of weird hardware configurations

2 marks for good disadvantages. E.g:

- Cannot control the testing
- Cannot guarantee anything about how thoroughly the software was tested
- Competitors may get hold of your software quicker
- Cannot guarantee the beta testers will report all errors they find
- Beta testers will report all sorts of things that are not errors