Software engineering issues

David Notkin
Autumn Quarter 2008
So far…

- ...design
- ...testing

- Today: *very limited* views of these two issues
  - Each is deserving of (at least) a course on its own
  - There are numerous other issues in software engineering including requirements and specification, analysis, maintenance, etc.
Design

- What goes in the scanner vs. what goes in the parser?
- How to decide?
Possible answers include…

- Cohesion – why are elements placed together into components?
  - “component” is intentionally pretty vague here, and could include packages, classes, modules, etc.
- Coupling – what are the interconnections and dependences between components (and why)?
- Anticipating change – what are likely changes and how will they be accommodated?
- Simplicity – see Hoare’s quotation, next slide
- Conceptual integrity – is there a consistent approach to existing decisions?
- … others?
Hoare sez

- “There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.”
Software structure degrades

• There is plenty of evidence that software structure degrades over time
• That is, well-planned and well-designed software systems become increasingly tangled over time
  – Less simple, less clear cohesion, more muddled coupling, harder to change, etc.
• One reason for this is that programmers often change code in a way that is locally sensible but has poor global and long-term consequences
• Reducing the rate of increase in entropy generally demands more global knowledge of the software
MiniJava

• As much as possible, respect the existing design – that is, try to maintain its conceptual integrity

• At least two reasons
  – Chambers, who wrote it originally, is a top-notch designer and programmer
  – You will end up with fewer unexpected interactions and problems
Software testing

• What are possible goals of software testing?
Dijkstra

- “Testing can only be used to show the presence of bugs, not their absence.”
What are alternatives to these goals?

• Formal verification of the software
  – Verification vs. validation: Building the system right vs. building the right system [Boehm]
• Inspections, reviews, walkthroughs
• Certifying the process (e.g., ISO9000)
• Certifying the practitioners (e.g., licensing doctors)
• …
A broad-brush of some testing issues

- White-box vs. black-box testing
  - Can see the code, can’t see the code
- Functional vs. performance vs. stress vs. acceptance vs. beta vs. … testing
- Structural coverage testing
Some terminology

- A **failure** occurs when a program doesn’t satisfy its specification.
- A **fault** occurs when a program’s internal state is inconsistent with what is expected (this is usually an informal notion).
- A **defect** is the code that leads to a fault (and perhaps a failure).
- An **error** is the mistake the programmer made in creating the defect.
A simple problem

- The program reads three integer values. The three values are interpreted as representing the lengths of the sides of a triangle. The program prints a message that states whether the triangle is isosceles, equilateral or scalene.
- Write a set of test cases that would adequately test this program
A study showed...

- 13 kinds of defects were found in actual programs
- Experienced programmers on average write test cases that identify about half of the defects
The lucky thirteen

• Valid scalene
• Valid equilateral
• Valid isosceles
• All permutations that represent valid scalene
• One side is zero
• One side is negative
• All sides are zero

• Three positive integers where two sum to the third
• All permutations of the previous case
• Three positive integers where two sum to less than the third
• All permutations of this
• A non-integer side
• An incorrect number of inputs
Bach adds…

• A GUI that accepts the three inputs
• Asks his students to “try long inputs”
• Interesting lengths
  – 16 digits+: loss of mathematical precision
  – 23+: can’t see all of the input
  – 310+: input not understood as a number
  – 1000+: exponentially increasing freeze when navigating to the end of the field by pressing <END>
  – 23,829+: all text in field turns white
  – 2,400,000: reproducible crash
• The programmer was only aware of the first two boundaries
“What stops testers from trying longer inputs?”

• Bach suggests
  – Seduced by what’s visible
  – Think they need the specification to tell them the maximum – and if they have one, stop there
  – Satisfied by first boundary
  – Use linear lengthening strategy
  – Think “no one would do that”
  – …
Partition testing

• Basic idea: divide program input space into (quasi-)equivalence classes, selecting at least one test case from each class
Structural coverage testing

• Premise: if significant parts of the program structure are not tested, testing is surely inadequate
• Control flow coverage criteria
  – Statement (node, basic block) coverage
  – Branch (edge) and condition coverage
  – Data flow (syntactic dependency) coverage
  – Others…
• Attempted compromise between the impossible and the inadequate
Statement coverage

• What’s a statement?
  – `max = (x > y) ? x : y;`
  – Using basic blocks can help this issue

• Obviously unsatisfying in trivial cases (such as the second example on the right, from Ghezzi)

```plaintext
if x > y then
  max := x
else
  max := y
endif

if x < 0 then
  x := -x
endif
z := x;
```
Edge coverage

• Uses control flow graph
  – We’ll see these soon!
  – Essentially a flowchart

• Covering all basic blocks (nodes) would not require edge ac to be covered

• Edge coverage requires all control flow graph edges to be coverage by at least one test
Condition coverage

• How to handle compound conditions?
  – if (p != NULL) && (p->left < p->right) ...

• Is this a single conditional in the CFG?

• How do you handle short-circuit conditionals?
  – andthen, orelse ...

• Condition coverage treats these as separate conditions and requires tests that handle all combinations
Path coverage

• Edge coverage is in some sense very static
• Edges can be covered without covering actual paths (sequences of edges) that the program may execute
• Note that not all paths in a program are always executable
  – Writing tests for these is hard 😊
  – Not shipping a program until these paths are executed does not provide a competitive advantage 😊
• Loops (or recursion) makes life even harder
Summary

• Software testing – and only parts were covered at the lightest imaginable level – is a complex art.
• But you need to be able to wear two hats – that of the developer, and that of the tester – and this is extremely hard.
• These ideas may give you some more disciplined way to think about your testing process, informal though it will be.