Testing:
Who\textsubscript{3}, What\textsubscript{4}, Why\textsubscript{1}, When\textsubscript{2}, How\textsubscript{5}

Lian Yu, Peking U.
Michal Young, U. Oregon

Why Test

- Stupid question?
  - But we need to be clear about goals before we can make reasoned choices regarding the other questions, *who, what, when, and how*

- We test to avoid costs
  - Cost of defects in the final product
  - Costs during software development

What about measurement?

- We can also test to measure the quality of software
  - Is it good enough to release?
  - Should we fix it or replace it?

- Different goals, different approaches
  - (put this aside for the moment ...)

Errors, Detection, and Repairs

- Basic observation:
  - Cost of a defect grows *quickly* with time between making an error and fixing it
    - Step function as defects cross scope walls: From programmer to sub-team, from close colleagues to larger team, from module to system, from developers to independent testers and from development to production

- "Early" errors are the most costly
  - Misunderstanding of requirements, architecture that does not support a needed change, ...
When

- As early as possible
  - Reduce the gap between making an error and fixing it
    - Ideally to “immediate” ... which we call “prevention” or “syntactic checking”, and involves deep changes in process and notations
- Throughout development
  - People make mistakes in every activity, so every work product must be tested as soon as possible

What

- Narrow view:
  - Testing is executing a program and comparing actual results to expected results
- Wider view:
  - “Testing” is shorthand for a variety of activities: anything we can do to check for defects.
  - Dynamic program testing is the most common activity when the artifact is program code
  - Other activities include reviews, analysis of models, automated checks. We usually need several.

Choosing What

- For every work product, we ask: How can I find defects as early as possible
  - Ex: How can I find defects in software architecture before we’ve designed all the modules? How can I find defects in my module code before it’s integrated into the system?
- Divide and conquer
  - What properties can be checked automatically?
  - What properties can be (effectively) tested dynamically?
  - How can I make reviews cost-effective?

Verification and Validation: Divide and Conquer

- Validation vs Verification
  - Are we building the right product? Are we building it right?
  - Crossing from judgment to precise, checkable correctness property. Verification is at least partly automatable, validation is not
- Correctness is a relation between spec and implementation
  - To make a property verifiable (testable, checkable, ...), we must capture the property in a spec
Divide and Conquer: Usability

- Real requirement:
  - The product must be usable. Users with characteristics XXX should learn to use it effectively within 30 minutes, and should thereafter complete task T within S seconds with error rate E.
  - Hard and expensive (but important) to test. We probably can’t test it after every trivial change to the product.
- Divide and conquer:
  - Validate the user interface design. Verify the user interface implementation: Is it consistent with the design? Does it validate any of the (precisely stated!) guidelines?

Testing: Who (from Why and When)

- Different goals, different actors
  - Usability example: Validating the UI design is an expensive activity requiring expertise in usability, best carried out in a usability lab. Verifying compliance to UI design should be simple and mostly automatable, carried out by developers and development testers. Measuring reliability is yet another goal, best carried out by an independent team.
- When: Now
  - Timeliness often constrains responsibility, e.g., unit testing is carried out by developers (but perhaps reviewed by others)

Who (in a distributed team)

- Cost of a defect rises dramatically at architectural and sub-team boundaries
  - It’s cheap for me to fix the bug I just created in my module. It’s much, much more expensive to find, understand, and fix a bug in a module made by my teammate who is sleeping 3000 miles away.
- => Test cases are part of good module interface designs
- => Module tests should be thorough and completed before a module (or revision) becomes part of the baseline used by others

The Long When

- Test execution is just one part of testing
  - And it needs to be a very cheap, automated part, because we should re-test the program over and over as it evolves
- Test design can often be done much earlier
  - Example: Part of a good system design is devising acceptance test cases.
- Test design is also a test of specifications
  - Is this specification precise, or ambiguous? Can I effectively check whether an implementation satisfies it?
Example: Test design from (module interface) specification

• Method: Compute vertical stripes
  » From a set of user-drawn rectangles on the display, return a list of shrinkable stripes such that:
    • Each stripe is entirely within the range of x values in the map
    • No two vertical stripes are adjacent
    • For any point x within the bounds of the map, either x is between the low and high values of one of the rectangles, or x is within one of the stripes, but not both.
  • Signature: List<intPair>
    getVStripes(Rect mapArea, List<Rect>)
• Test cases?

How (from why, who, when, what)

• Black box: Test design is part of designing good specifications
  • This will change specs, in a good way. Factoring validation from verification is particularly hard, but particularly cost-effective as it leverages and focuses expensive human judgment
• White (or glass) box: Test design from program design
  • Executing every statement or branch does not guarantee good tests, but omitting a statement is a bad smell.

More!

• Testing and process refinement:
  • Effective organizations track and classify defects. What do we typically get wrong? Which bugs cost us most? What causes them? How can we cost-effectively prevent or detect them?
• Testing and incentive structures:
  • It’s easy to accidentally create perverse incentives, e.g., to turn in modules on schedule with latent bugs. If the incentive structure is broken, nothing else will work.
• Review techniques, automatization, ...

Plug

SOFTWARE TESTING AND ANALYSIS
MAURO PEZZÈ
MICHAEL YOUNG