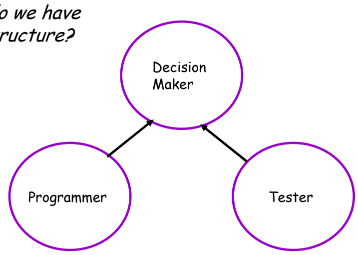
Testing:
Methods, Practice, Research

State of the World

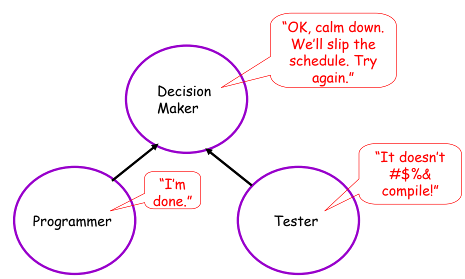
- Standard software development is simple
  - No rocket science here
- Outline
  - Someone writes a program
  - Someone runs the program and checks that it behaves as expected
  - Someone decides when it is OK to release

Software Development Today

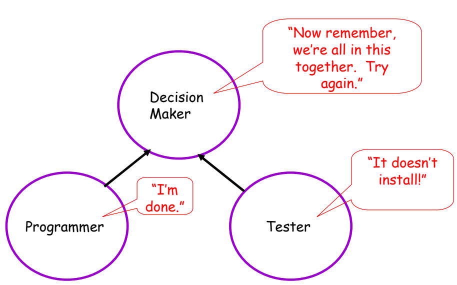
Why do we have this structure?



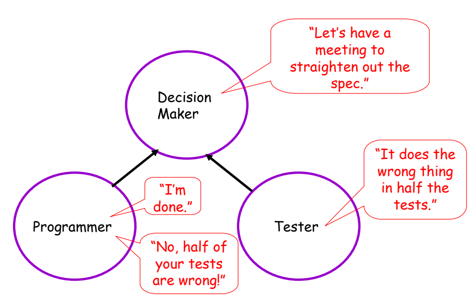
Typical Scenario (1)



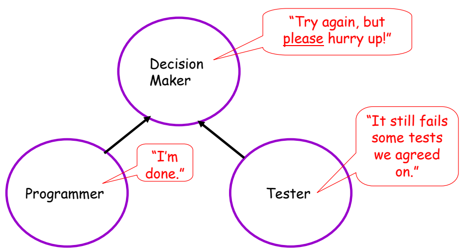
Typical Scenario (2)



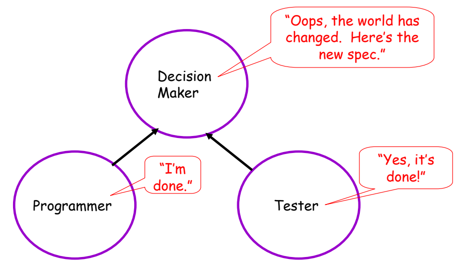
Typical Scenario (3)



Typical Scenario (4)



Typical Scenario (5)



Key Assumptions

- Development and testing must be independent
- Specifications must be explicit
- Specifications are always evolving
- All resources (including time) are finite
- Human organizations need decision makers

- Examine each of these separately

Independent Testing and Development

- Testing is basic to every engineering discipline
  - Design a drug
  - Manufacture an airplane
  - Etc.

- Why?
  - Because our ability to predict how our creations will behave is imperfect
  - We need to check our work, because we will make mistakes

Independent Testing and Development of Software

- In what way is software different?

- Two aspects:
  - Folklore: "Programmers are optimists"
  - The implication is that programmers make poor testers

  - Economics: "Programming costs more than testing"
  - The implication is that programming is a higher-skill profession

- How valid is the folklore, and how much is due to the current state of the art in testing?

Explicit Specifications

- Software involves multiple people
  - At least a programmer and a user
  - But usually multiple programmers, testers, etc.

- Any team effort requires mutual understanding of the goal
  - A specification
  - Otherwise, team members inevitably have different goals in mind

Specifications Change

- Why?
- Many software systems are truly "new"
  - Differ from all that went before in some way
  - Initial specification will change as problems are discovered and solved
- The world is changing
  - What people want
  - The components you build on (e.g., the OS version)

Software Specifications

- Software specifications are usually
  - in prose
  - imprecise
  - out of date
- Current state of specification is not conducive to automation
  - Not consumable by tools
  - Without a specification, there is nothing to check

Finite Resources

- Organizations make trade-offs
  - Not all goals can be achieved
  - Because resources are finite
- $'s express relative costs among goals
  - Goals that are hard to quantify pose a problem
  - E.g., correctness, completeness

"We have 2 months, 5 programmers, and 2 testers. Here is a priority list of features. A feature is finished when it passes all of the tests for that feature; a programmer does not move on to a new feature until all higher priority features are finished or assigned to other programmers. We start now and ship whatever features are finished in 60 days."

Summary of the State of the World

- Software development today relies overwhelmingly on the coder/tester model
- Typically half of the expense in developing a software product is in testing
  - And overwhelming, this testing is low tech

Some Testing Topics

- Industry practices
  - Code coverage
  - Black-box and white-box testing
  - State-of-the-art commercial tools
- Testing theory
  - Hardness results, testing finite state machines
- Research problems in testing
  - E.g., fault injection

Dynamic Analysis Topics (Preliminary)

- Efficient tracing
- Code instrumentation
- Deriving invariants from traces
- Monitoring long-running systems
- Commercial tools
  - E.g., Purify

Specifications

- Specifications are needed for *any* technique
  - Why? Because no tool can divine what the software is supposed to do.
- Every method is a variation on:
  - Get people to say something in two different ways
  - Check the two versions for consistency
    - E.g., variables' types and their actual usage
    - E.g., test cases and the compiled code

Specifications (Cont.)

- *Every* technique relies on specifications
  - If only the semantics of the language
- The current state of specification is poor
- How can we get more specifications into programs?
  - Partial specs
  - Lightweight specs

Testing Practice

Reality

- Researchers have investigated many approaches to improving software quality
- But the world tests
- > 50% of the cost of software development is testing
- Testing is important

Testing Topics

- Purpose of testing
- Widely-used practices
  - Manual testing
  - Automated testing
  - Regression testing
  - Nightly build
  - Code coverage
  - Bug trends
  - Stress testing

The Purpose of Testing

Two purposes:

1. Find bugs
   - Find important bugs
2. Elucidate the specification

Example

- Test case
  Add a child to Mary Brown's record

- Version 1
  - Check that Ms. Brown's # of children is one more
- Version 2
  - Also check Mr. Brown's # of children
- Version 3
  - Check that no one else's child counts changed

Specifications

- Good testers clarify the specification
  - This is creative, hard work
  - There is no realistic hope that tools will ever automate this

- We bemoan the lack of specifications in software

- But testers *are* creating specifications

Manual Testing

- Test cases are lists of instructions
  - "test scripts"

- Someone manually executes the script
  - Do each action, step-by-step
    - Click on "login"
    - Enter username and password
    - Click "OK"
    - …
  - And manually records results

- Low-tech, simple to implement

Manual Testing

- Manual testing is very widespread
  - Probably not dominant, but very, very common
- Why? Because
  - Some tests can't be automated
    - Usability testing
  - Some tests shouldn't be automated
    - Not worth the cost
- There are also not-so-good reasons
  - Not-so-good because innovation could remove them
  - Testers aren't skilled enough to handle automation
  - Automation tools are too hard to use

Automated Testing

- Idea:
  - Record manual test
  - Play back on demand

- This doesn't work as well as expected

Fragility

- Test recording is usually very fragile
  - Breaks if environment changes anything
  - E.g., location, background color of textbox

- More generally, automation tools cannot generalize a test
  - They literally record exactly what happened
  - If anything changes, the test breaks

- A hidden strength of manual testing
  - Because people are doing the tests, ability to adapt tests to slightly modified situations is built-in

Breaking Tests

- When code evolves, tests break
  - E.g., change the name of a dialog box
  - Any test that depends on the name of that box breaks
- Maintaining tests is a lot of work
  - Broken tests must be fixed; this is expensive
  - Cost is proportional to the number of tests
  - Implies that more tests is not necessarily better

Improved Automated Testing

- Recorded tests are too low level
  - E.g., every test contains the name of the dialog box
- Need to abstract tests
  - Replace dialog box string by variable name X
  - Variable name X is maintained in one place
    - So that when the dialog box name changes, only X needs to be updated and all the tests work again
- This is just structured programming
  - Just as hard as any other system design
  - Really, a way of making the specification more concise

Back to Specifications

- Specifying software is really hard
- In formal methods community, much bemoaning of level of detail required to specify a system
  - But this has *nothing* to do with formal methods
  - Any specification approach must express the details
- The difficulty of automating testing is in the same category

Discussion

- Testers have two jobs
  - Clarify the specification
  - Find (important) bugs
- Only the latter is subject to automation
- Helps explain why there is so much manual testing

Regression Testing

- Idea
  - When you find a bug,
  - Write a test that exhibits the bug,
  - And always run that test when the code changes,
  - So that the bug doesn't reappear
- Without regression testing, it is surprising how often old bugs reoccur

Regression Testing (Cont.)

- Regression testing ensures forward progress
  - We never go back to old bugs
- Regression testing can be manual or automatic
  - Ideally, run regressions after every change
  - To detect problems as quickly as possible
- But, regression testing is expensive
  - Limits how often it can be run in practice
  - Reducing cost is a long-standing research problem

Regression Testing (Cont.)

- Note other tests (besides bug tests) can be checked for regression
- Ideally, entire suite of tests is rerun on a regular basis to assure old tests still work

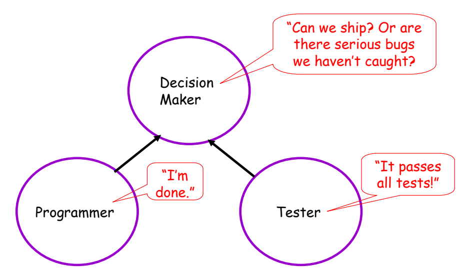
Nightly Build

- Build and test the system regularly
  - Every night
- Why? Because it is easier to fix problems earlier than later
  - Easier to find the cause after one change than after 1,000 changes
  - Avoids new code from building on the buggy code
- Test is usually subset of full regression test
  - “smoke test”
  - Just make sure there is nothing horribly wrong

A Problem

- So far we have:
  - *Measure changes regularly* (nightly build)
  - *Make monotonic progress* (regression)
- How do we know when we are done?
  - Could keep going forever
- But, testing can only find bugs, not prove their absence
  - We need a proxy for the absence of bugs

Typical Scenario



Code Coverage

- Idea
  - Code that has never been executed likely has bugs
- This leads to the notion of *code coverage*
  - Divide a program into units (e.g., statements)
  - Define the coverage of a test suite to be

$$\frac{\text{\# of statements executed by suite}}{\text{\# of statements}}$$

Code Coverage (Cont.)

- Code coverage has proven value
  - It’s a real metric, though far from perfect
- But 100% coverage does not mean no bugs
  - E.g., a bug visible after loop executes 1,025 times
- And 100% coverage is almost never achieved
  - Ships happen with < 60% coverage
  - High coverage may not even be desirable
    - May be better to devote more time to tricky parts with good coverage

Using Code Coverage

- Code coverage helps identify weak test suites
- Tricky bits with low coverage are a danger sign
- Areas with low coverage suggest something is missing in the test suite

Example

```
status = perform_operation();
if (status == FATAL_ERROR)
  exit(3);
```

- Coverage says the exit is never taken
- Straightforward to fix
  - Add a case with a fatal error
- But are there other error conditions that are not checked in the code?

The Lesson

- Code coverage can't complain about missing code
  - The case not handled
- But coverage can hint at missing cases
  - Areas of poor coverage ⇒ areas where not enough thought has been given to specification

Bug Trends

- Idea: Measure rate at which new bugs are found
  - Rational: When this flattens out it means
    1. The cost/bug found is increasing dramatically
    2. There aren't many bugs left to find
- Assumes testing resources are well-deployed
  - We aren't overlooking any part of the code
- Assumes bugs can be fixed

Stress Testing

- Push system into extreme situations
  - And see if it still works…
- Stress
  - Performance
    - Feed data at very high or very low rates
  - Interfaces
    - Replace APIs with badly behaved stubs
  - Internal structures
    - Works for any size array? Try sizes 0 and 1
  - Resources
    - Set memory artificially low
    - Same for # of file descriptors, network connections, etc.

Stress Testing (Cont.)

- Stress testing will find many obscure bugs
  - Explores the corner cases of the design
- Some may not be worth fixing
  - As unlikely in practice
- A corner case now is tomorrow's common case
  - Data races, data sizes always increasing
  - Software is often stress tested

The Big Picture

- Testing practice has grown by trial-and-error
  - Many, many errors
- Standard practice
  - *Measure progress often* (nightly builds)
  - *Make forward progress* (regression testing)
  - *Stopping condition* (coverage, bug trends)

What Can We Learn From Testing Research?

- Observations
  - A huge amount of labor goes into testing
    - > 50% of project investment
  - Much of this labor just ferrets out the spec
- Question: Can we redirect this effort into more useful specifications?
  - More useful for tools, that is

Testing Research

Overview

- Testing research has a long history
  - At least to the 1960's
- Much work is focused on metrics
  - Assigning numbers to programs
  - Assigning numbers to test suites
  - Heavily influenced by industry practice
- More recent work focuses on deeper analysis
  - Semantic analysis, in the sense we understand it

Random Testing

- About $\frac{1}{4}$ of Unix utilities crash when fed random input strings
  - Up to 100,000 characters
- What does this say about testing?
- What does this say about Unix?

What it Says About Testing

- Randomization is a highly effective technique
  - And we use very little of it in software
- "A random walk through the state space"
- To say anything rigorous, must be able to characterize the distribution of inputs
  - Easy for string utilities
  - Harder for systems with more arcane input
    - E.g., parsers for context-free grammars

What it Says About Unix

- What sort of bugs did they find?
  - Buffer overruns
  - Format string errors
  - Wild pointers/array out of bounds
  - Signed/unsigned characters
  - Failure to handle return codes
  - Race conditions
- Nearly all of these are problems with C!
  - Would disappear in Java
  - Exceptions are races & return codes

One Interesting Bug

csh !0%8f

- ! is the history lookup operator
  - No command beginning with 0%8f
- csh passes an error "0%8f: Not found" to an error printing routine
- Which prints it with printf()

Efficient Regression Testing

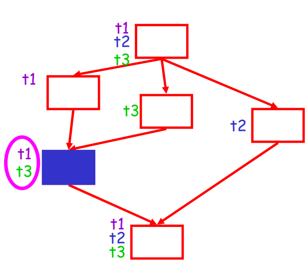
- Problem: Regression testing is expensive
- Observation: Changes don't affect every test
  - And tests that couldn't change need not be run
- Idea: Use a conservative static analysis to prune test suite

The Algorithm

Two pieces:

1. Run the tests and record for each basic block which tests reach that block
2. After modifications, do a DFS of the new control flow graph. Wherever it differs from the original control flow graph, run all tests that reach that point

Example



Label each node of the control flow graph with the set of tests that reach it.

When a statement is modified, rerun just the tests reaching that statement.

Experience

- This works
  - And it works better on larger programs
  - # of test cases to rerun reduced by > 90%
- Total cost less than cost of running all tests
  - Total cost = cost of tests run + cost of tool
- Why not use this?

What is a Good Test?

- We're implementing a function F on domain D
- A test set $T \subseteq D$ is *reliable* if for all programs P

$$(\forall t \in T.\ P(t) = F(t)) \Rightarrow (\forall t \in D.\ P(t) = F(t))$$

- Says that a good test set is one that implies the program meets its specification

Good News/Bad News

- Good News
  - There are interesting examples of reliable test sets
  - Example: *A function that sorts N numbers using comparisons sorts correctly iff it sorts all inputs consisting of 0,1 correctly*
  - This is a finite reliable test set
- Bad News
  - There is no effective method for generating finite reliable test sets

An Aside

- It's clear that reliable test sets must be impossible to compute in general
- But most programs are not diagonalizing Turing machines…
- It ought to be possible to characterize finite reliable test sets for certain classes of programs

What is a Good Test?

- We're implementing a function F on domain D
- A test set $T \subseteq D$ is *reliable* if for all programs P

  $(\forall t \in T.\ P(t) = F(t)) \Rightarrow (\forall t \in D.\ P(t) = F(t))$
- equivalently, for all programs P

  $(\exists t \in D.\ P(t) \neq F(t)) \Rightarrow (\exists t \in T.\ P(t) \neq F(t))$
- But we can't afford to quantify over all programs . . .

From Infinite to Finite

- We need to cut down the size of the problem
  - Check reliability w.r.t. a smaller set of programs
- Idea: Just check a finite number of (systematic) variations on the program
  - E.g., replace x > 0 by x < 0
  - Replace I by I+1, I-1
- This is *mutation analysis*

Mutation Analysis

- Modify (mutate) each statement in the program in finitely many different ways
- Each modification is one *mutant*
- Check for adequacy w.r.t. the set of mutants
  - Find a set of test cases that distinguishes the program from the mutants

What Justifies This?

- The "competent programmer assumption"
  The program is close to right to begin with
- It makes the infinite finite
  We will inevitably do this anyway; at least here it is clear what we are doing

The Plan

- Generate mutants of program P
- Generate tests
  - By some process
- For each test t
  - For each mutant M
    - If M(t) ≠ P(t) mark M as *killed*
- If the tests kill all mutants, the tests are reliable

The Problem

- This is dreadfully slow
- Lots of mutants
- Lots of tests
- Running each mutant on each test is expensive
- But early efforts more or less did exactly this

Better Algorithms

- Observation: Mutants are nearly the same as the original program
- Idea: Compile one program that incorporates and checks all of the mutations simultaneously
  - A so-called *meta-mutant*
- Weak mutation
  - Check only that mutant produces different state after mutation, not different final output

Metamutant with Weak Mutation

- Constructing a metamutant for weak mutation is straightforward
- A statement has a set of mutated statements
  - With any updates done to fresh variables
    $X := Y \ll 1 \qquad X_1 := Y \ll 2 \quad X_2 := Y \gg 1$
  - After statement, check to see if values differ
    $X == X_1 \qquad X == X_2$

Comments

- A metamutant for weak mutation should be quite practical
  - Constant factor slowdown over original program
- If test suite fails to kill all mutants, then (maybe) it is inadequate