

Testing

17-654/17-754

Analysis of Software Artifacts

Jonathan Aldrich



Analysis of Software Artifacts -
Spring 2006

Outline



- What is testing?
 - Goals and nature of testing
 - Place in quality assurance strategy
- How to test?
 - Kinds of testing
 - Important techniques
 - Effective testing practices
- When to test?
 - Software lifecycle and process
- When to stop?
 - Testing metrics
 - How to decide when you're done

Analysis of Software Artifacts -
Spring 2006

2

What is Testing?



- Direct execution of code on test data in a controlled environment [Scherlis]

Goals of Testing?



Goals of Testing



- To reveal failures
 - Most important goal of testing
- To measure quality
 - Difficult to quantify, but still important
- To clarify the specification
 - Always test with respect to a spec
 - Testing shows inconsistency
 - Either spec or program could be wrong
- To learn about program
 - How does it behave under various conditions?
 - Feedback to rest of team goes beyond bugs
- To verify contract
 - Includes customer, legal, standards

Testing is **NOT** to Show Correctness



- Unrealistic
 - The program is not correct!
- Counterproductive
 - Bad tester psychology
 - You fail when program does
 - Psychology experiment
 - People look for blips on screen
 - They notice more if rewarded for finding blips than if penalized for giving false alarms
 - Testing for bugs is more successful than testing for correctness
 - Teasley, Leventhal, Mynatt & Rohlman: Why software testing is sometimes ineffective: Two applied studies of positive testing strategy

Complete Testing



- **Definition**
 - At the end of testing, you know there are no remaining unknown bugs
 - [Kaner, Bach]
- **Impossible!** (for non-trivial programs & specs)
 - **Proof by contradiction**
 - Assume you have tested a program completely
 - If the program is non-trivial, there is untested input
 - If the spec is non-trivial, there is a way the program can fail on that input
 - if (input == aUntestedCase)
 then exhibitBug()
 else runCorrectProgram()

Tough Bugs [Kaner et al.]



- When entering data, a user fidgets, pressing alternatively number and backspace. When the number is finally entered, all the numbers and backspace keys are flushed from a buffer to the server, overflowing the input buffer and crashing the system.
- A database management program breaks with files a multiple of 16,384 bytes long
- A word processor deleting paragraphs from large, fragmented disk files during editing
- A telephone system has 6 states, one of which involves placing a caller on hold. The system keeps a stack of calls placed on hold. If the caller hangs up while on hold, the information is left on the stack until the phone is idle. But if 30 callers hang up before the phone is next idle, the stack overflows and the phone crashes.

Testing Terminology



- Failure
 - Program does not meet its specification
- Fault
 - Internal state is inconsistent with expectation
 - May leads to failure
- Defect
 - Code that causes a fault
 - May lead to a failure

Testing for Quality Assurance



- One technique among many
 - Testing
 - Assertions
 - Inspection
 - Static analysis (Fluid)
 - Types (Java)
 - Model checking (Blast)
 - Theorem proving (ESC/Java)
- Testing is primary for most organizations
 - Often more cost-effective than inspection
 - Static analysis does not apply to all attributes
 - Or may be prohibitively expensive, e.g. ESC/Java
- Costly
 - May be more expensive than development
 - Improvement is critical
 - Better quality
 - Same quality at lower cost

Which Technique for Which Attribute?



- Correct output
- Performance/scaleability
- Null dereferences
- Encapsulation
- Security
- Protocol compliance
- Standards conformance
- Concurrency
- Memory errors
- Usability
- Evolvability
- Localization

Cost/Benefit Tradeoffs



- Static analysis
 - Benefit: can eliminate errors
 - Cost varies enormously, but low in well-designed, mature system because of automation
- Testing
 - Benefit: can check almost anything, but unsound
 - Cost: medium
- Inspections
 - Benefit: can check what nothing else can
 - Certain security attributes
 - Evolvability
 - May find other errors missed by testing
 - Cost: probably the highest of all

When is one test more valuable than others?



Test Case Value



- Value is driven by quality improvement
 - Some value of information as well
- Value Factors
 - Does it find a bug?
 - How severe is the bug?
 - How common is the bug?
 - How easy is it to fix the bug?
 - Is it distinct from other tests?
 - Unique bug? Unique code? Unique domain coverage?
 - How general is it?
 - What did we learn about the program?

How to test?



- Kinds of testing
- Important techniques
- Effective testing practices

Testing Exercise [Kaner et al.]



- The program is designed to add two numbers, which you enter. Each number should be one or two digits. The program will echo your entries, then print the sum. Press <Enter> after each number.

Testing Quality Attributes



- Performance
- Reliability
- Fault tolerance
- Security
- Usability
- Portability
- Evolvability

Outline



- What is testing?
 - Goals and nature of testing
 - Place in quality assurance strategy
- **How to test?**
 - **Kinds of testing**
 - Important techniques
 - Effective testing practices
- When to test?
 - Software lifecycle and process
- When to stop?
 - Testing metrics
 - How to decide when you're done

White-Box (Glass-Box, Structural) Testing

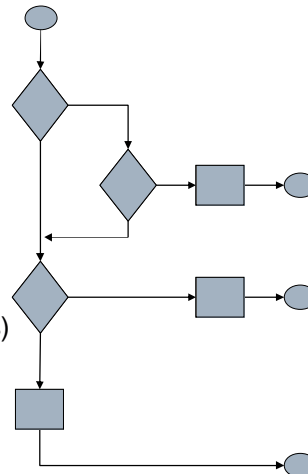


- Look at the code (white-box) and try to systematically cause it to fail
- Coverage criteria: a way to be systematic
 - E.g. cover all statements in the program
- Is coverage realistic?
 - Why might you not be able to achieve 100% coverage?

Statement Coverage [Slide from Bill Scherlis]



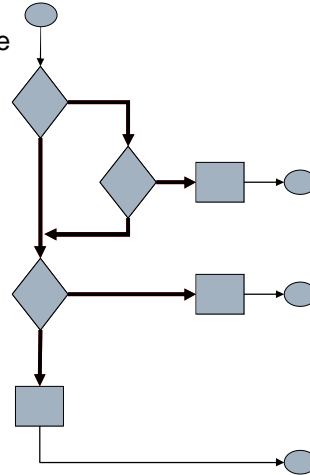
- Statement coverage
 - What portion of program statements (nodes) are touched by test cases
- Advantages
 - Test suite size linear in size of code
 - Coverage easily assessed
- Issues
 - Dead code is not reached
 - May require some sophistication to select input sets (McCabe basis paths)
 - Fault-tolerant error-handling code may be difficult to “touch”
 - Metric: Could create incentive to *remove* error handlers!



Branch Coverage [Slide from Bill Scherlis]



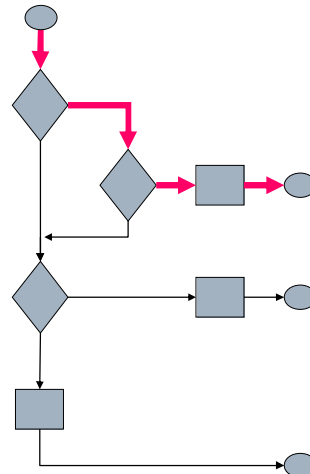
- **Branch coverage**
 - What portion of condition branches are covered by test cases?
 - Or: What portion of relational expressions and values are covered by test cases?
 - Condition testing (Tai)
- **Advantages**
 - Test suite size and content derived from structure of boolean expressions
 - Coverage easily assessed
- **Issues**
 - Dead code is not reached
 - Fault-tolerant error-handling code may be difficult to “touch”



Path Coverage [Slide from Bill Scherlis]



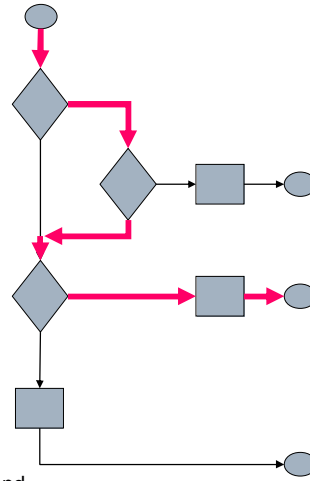
- **Path coverage**
 - What portion of all possible paths through the program are covered by tests?
 - Loop testing: Consider representative and edge cases:
 - Zero, one, two iterations
 - If there is a bound n : $n-1$, n , $n+1$ iterations
 - Nested loops/conditionals from inside out
- **Advantages**
 - Better coverage of logical flows
- **Disadvantages**
 - Not all paths are possible, or necessary
 - What are the *significant* paths?
 - Combinatorial explosion in cases unless careful choices are made
 - E.g., sequence of n if tests can yield up to 2^n possible paths
 - Assumption that program structure is basically sound



Path Coverage [Slide from Bill Scherlis]



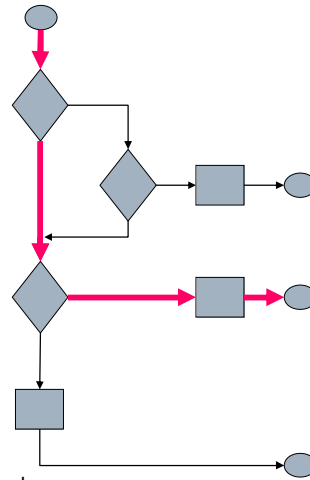
- Path coverage
 - What portion of all possible paths through the program are covered by tests?
 - Loop testing: Consider representative and edge cases:
 - Zero, one, two iterations
 - If there is a bound n : $n-1$, n , $n+1$ iterations
 - Nested loops/conditionals from inside out
- Advantages
 - Better coverage of logical flows
- Disadvantages
 - Not all paths are possible, or necessary
 - What are the *significant* paths?
 - Combinatorial explosion in cases unless careful choices are made
 - E.g., sequence of n if tests can yield up to 2^n possible paths
 - Assumption that program structure is basically sound



Path Coverage [Slide from Bill Scherlis]



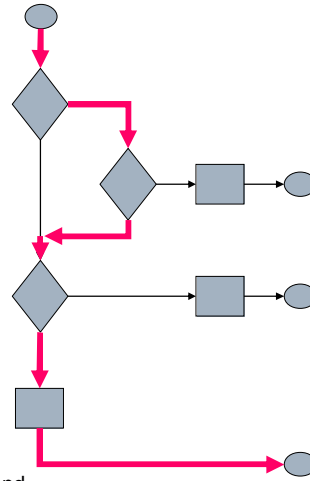
- Path coverage
 - What portion of all possible paths through the program are covered by tests?
 - Loop testing: Consider representative and edge cases:
 - Zero, one, two iterations
 - If there is a bound n : $n-1$, n , $n+1$ iterations
 - Nested loops/conditionals from inside out
- Advantages
 - Better coverage of logical flows
- Disadvantages
 - Not all paths are possible, or necessary
 - What are the *significant* paths?
 - Combinatorial explosion in cases unless careful choices are made
 - E.g., sequence of n if tests can yield up to 2^n possible paths
 - Assumption that program structure is basically sound



Path Coverage [Slide from Bill Scherlis]



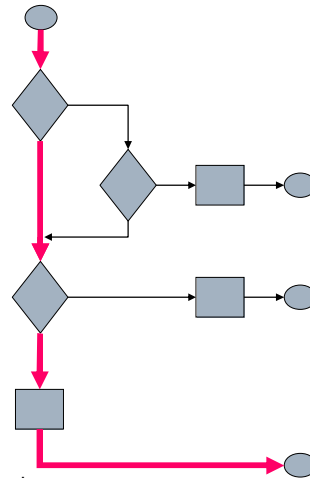
- Path coverage
 - What portion of all possible paths through the program are covered by tests?
 - Loop testing: Consider representative and edge cases:
 - Zero, one, two iterations
 - If there is a bound n : $n-1$, n , $n+1$ iterations
 - Nested loops/conditionals from inside out
- Advantages
 - Better coverage of logical flows
- Disadvantages
 - Not all paths are possible, or necessary
 - What are the *significant* paths?
 - Combinatorial explosion in cases unless careful choices are made
 - E.g., sequence of n if tests can yield up to 2^n possible paths
 - Assumption that program structure is basically sound



Path Coverage [Slide from Bill Scherlis]



- Path coverage
 - What portion of all possible paths through the program are covered by tests?
 - Loop testing: Consider representative and edge cases:
 - Zero, one, two iterations
 - If there is a bound n : $n-1$, n , $n+1$ iterations
 - Nested loops/conditionals from inside out
- Advantages
 - Better coverage of logical flows
- Disadvantages
 - Not all paths are possible, or necessary
 - What are the *significant* paths?
 - Combinatorial explosion in cases unless careful choices are made
 - E.g., sequence of n if tests can yield up to 2^n possible paths
 - Assumption that program structure is basically sound



Does 100% Coverage Find All Errors?



- Can you write a program that has a bug, even it tested with 100% path coverage?

Other Coverage Criteria



- Condition coverage
 - All combinations of sub-conditions of if condition true & false
- Predicate coverage
 - Choose a set of predicates
 - Cover each statement with each combination of predicates
- Exercise data structures
 - Each conceptual state or sequence of states

Dangers of Coverage



- Coverage focus misses important bugs
 - Missing code
 - Incorrect boundary values
 - Timing problems
 - Configuration issues
 - Data/memory corruption bugs
 - Usability problems
 - Customer requirements issues
- Coverage is not a good adequacy criterion
 - Instead, use to find places where testing is *inadequate*

Benefits of White-Box



- Tool support can measure coverage
 - Helps to evaluate test suite (carefull!)
 - Can find untested code
- Can test program one part at a time
- Can consider code-related boundary conditions
 - If conditions
 - Boundaries of function input/output ranges
 - e.g. switch between algorithms at data size=100
- Assertion testing
 - Embeds part of specification in code
 - Checked on every execution, at every assert, instead of just at end of test

Black-Box (Functional) Testing



- Verify each piece of functionality of the system
 - Black-box: don't look at the code
 - More common in practice than white-box
- Benefit: finds bugs white-box doesn't
 - Think like a user, not a programmer
 - The programmer already checked the code!
 - Timing, unanticipated errors, UI, concurrency, configuration issues, performance, hardware failures
- Drawbacks
 - No insight into code structure
 - But good testers will guess anyway!

Unit Tests



- Test a small piece of source code
 - Typical: one function at a time
- Usually automated
 - jUnit is a popular framework
- Often specified by developer (XP)
 - Can be run before every checkin
- Purpose
 - Catch bugs early
 - Improve coverage
 - Validate internal interface/API designs
 - Catch bugs before code is written
 - Use test suites to guide implementation (XP)
 - Test components before client/service code is available

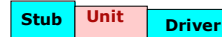
Unit Testing Principles



- Test anything that might fail
 - Unclear interface
 - Complicated implementation
 - Unusual case of usage
 - Defect found
 - About to refactor
- Don't test trivial methods
 - No benefit
 - Makes testing laborious

Simulate External Code with Scaffolding

[Scherlis]



- Client API
 - Model the software client for the service being tested
 - Create a **test driver**
 - Object-oriented approach:
 - Test individual calls and sequences of calls
- Service code
 - Underlying services
 - Communication services
 - Model behavior through a communications interface
 - Database queries and transactions
 - Network/web transactions
 - Device interfaces
 - Simulate device behavior and failure modes
 - File system
 - Create file data sets
 - Simulate file system corruption
 - Etc
 - Create a set of **stub** services or **mock** objects
 - *Minimal* representations of APIs for these services

Test Scaffolding [Scherlis]



- Purposes
 - Catch bugs early
 - Before client code or services are available
 - Limit the scope of debugging
 - Localize errors
 - Improve coverage
 - System-level tests may only cover 70% of code [Massol]
 - Simulate unusual error conditions – test internal robustness
 - Validate internal interface/API designs
 - Simulate clients in advance of their development
 - Simulate services in advance of their development
 - Capture developer intent (in the absence of specification documentation)
 - A test suite formally captures elements of design intent
 - Developer documentation
 - Enable division of effort
 - Separate development / testing of service and client
 - Improve low-level design
 - Early attention to ability to test – “testability”

Design by Contract



- General meaning
 - Specify a contract between client and implementation of a module
 - Using pre- and post-conditions
 - System works if both parties fulfill their contract
- Specific setting of testing
 - Verify pre- and post-conditions while running
 - Assign blame based on which one fails
 - Turns a system execution into a set of unit tests
- Supported by Eiffel, JML tools
 - Can simulate yourself with assert

Integration Testing



- Test several modules together
 - Still need scaffolding for modules not under test
- Avoid “big bang” integrations
 - Going directly from unit tests to whole program tests
 - Likely to have many big issues
 - Hard to identify which component causes each
- Test interactions between modules
 - Ultimately leads to end-to-end system test

Nightly Builds



- Building a release of a large project every night
 - Catches integration problems where a change “breaks the build”
 - Breaking the build is a BIG deal—may result in midnight calls to the responsible engineer
- Run “smoke test” on build
 - Tests basic functionality and stability of a build
 - Argue: anything that fails is in miserable shape
 - If it fails, do no further testing and continue with the last build while the issue is addressed
 - Ideally can be run by programmers before check-in

Regression Testing



- A suite of tests is run every time the system changes
- Goal: to catch any new bugs introduced by change
 - Need to add tests for new functionality
 - Always add tests that revealed a defect
 - If it happened once, it may happen again
 - But still test the old functionality also!
 - Note: in some cases, old test cases *should* return a different result, depending on the change that was made
- Very common for bug fixes to introduce new issues
 - Thus should plan several test-fix cycles

Test Automation Tradeoffs



- Costs
 - Automation: write once, run many times
 - Total cost = cost to develop + N * cost to run + cost to maintain
 - Cost to develop may be much higher relative to manual testing
 - When is it more expensive to test manually?
- Value of automated tests
 - Value = N * value per run
 - What is the incremental value of running a test again?

Test Automation Tradeoffs



- Costs
 - Automation: write once, run many times
 - Total cost = cost to develop + N * cost to run + cost to maintain
 - Cost to develop may be much higher relative to manual testing
 - When is it more expensive to test manually?
- Value of automated tests
 - Value = initial value + N * incremental value
 - Principle: a test is most valuable if you've never run it before!
 - Will you know if the test fails?
 - When is it worthwhile to pay cost of automation?
 - Pitfall: Design tests so they catch bugs, *not* so they are easy to automate!

Acceptance Tests



- Functional tests that the customer uses to evaluate the quality of the system
 - Should get explicit buy-in by customer: I will accept the code if these tests work

Release and Integrity Tests



- Release Test
 - Test release CD
 - Before manufacturing!
 - Check files for binary equality
 - Install and run
 - Virus scan CD and installation
- Integrity Test
 - Independent evaluation before release
 - Validate claims
 - Marketing, requirements, documentation
 - Anticipate product reviews, consumer complaints
 - Not really focused on bug-finding

Outline



- What is testing?
 - Goals and nature of testing
 - Place in quality assurance strategy
- How to test?
 - Kinds of testing
 - Important techniques
 - Effective testing practices
- When to test?
 - Software lifecycle and process
- When to stop?
 - Testing metrics
 - How to decide when you're done

Equivalence Class Testing



- Equivalence classes
 - A partition of a set
 - Usually the input domain of the program
 - Based on some equivalence relation
 - Intuition: all inputs in an equivalence class will fail or succeed in the same way

Equivalence Class Example



- Program Specification
 - Given 3 numbers, output whether a triangle formed from these number is equilateral, isosceles, or scalene
- Equivalence classes?

Finding Equivalence Classes



- Intuition that test cases are similar
 - This is useful, but can be incomplete
- Use cases in the specification
 - Impractical if you don't have the spec
 - What if the spec is incomplete?
- One class per code path
 - Impractical if you don't have code
- Risk-based
 - Consider a possible error as a risk
 - Given that error, what test cases will produce the same result?

Equivalence Class Hueristics



- Invalid inputs
- Ranges of numbers
- Membership in a group
- Equivalent outputs
 - Can you force the program to output an invalid or overflow value?
- Error messages
- Equivalent operating environments

What value to choose from Equiv Class?



- Risk-based: find the one most likely to find an error
- Boundary conditions
 - Common source of errors: off-by-one error at a boundary between equivalence classes

Boundary Conditions



- Intuition
 - Boundary conditions will usually find errors that are present in any other member of the equivalence class
 - But they may find off-by-one errors as well
- Robustness Testing
 - Test both just inside and just outside boundary
 - Ensures classification of input is accurate, and that appropriate error messages are given

Equivalence Class Tables



Variable	Valid ECs	Invalid ECs	Boundaries, Special cases	Notes
First Number				

Equivalence Class Tables



Variable	Valid ECs	Invalid ECs	Boundaries, Special cases	Notes
Triangle Input				

Risk-Based Equivalence Class Table

Kaner et al.



Variable	Risk (failure scenario)	Non-failure classes	Failure Classes	Test Cases (best representatives)
First Number				

Analysis of Software Artifacts -
Spring 2006

53

Combination Testing



- Some errors might be triggered only if two or more variables are at boundary values
- Test combinations of boundary values
 - Combinations of valid input
 - One invalid input at a time
 - In many cases no added value for multiple invalid inputs
- Subtlety required
 - What are the boundary cases for an application that deals with months and days?

Analysis of Software Artifacts -
Spring 2006

54

Outline



- What is testing?
 - Goals and nature of testing
 - Place in quality assurance strategy
- **How to test?**
 - Kinds of testing
 - Important techniques
 - **Effective testing practices**
- When to test?
 - Software lifecycle and process
- When to stop?
 - Testing metrics
 - How to decide when you're done

Testing Practices: Reporting Defects



- **Reproducibility**
 - Easier to fix the defect
 - Easier to validate a fix
 - Increased confidence that defect is real
- **Report as simple and general a defect as possible**
 - Easier to fix
 - More convincing problem,
 - Avoids overly narrow (symptom-focused) fix
- **Report in a non-antagonistic way**
 - Just state the problem
 - Don't blame or hypothesize about cause

Social Issues of Testing



- It's easy to shoot the messenger
 - Be objective, impersonal, and clear
- Don't use defects in performance evaluations
 - Endless argument about whether a bug is real
 - Testers will be asked to retract duplicate reports, reports with the same underlying problem, design suggestions, irreproducible reports
 - Bad will between testers and developers
 - Lawsuits
- Your experiences?

Testing Practices: Analyzing Defects



- Try to find multiple paths to a problem
 - If one path is common, defect is higher priority
 - Each path provides more info on likely cause
- Try to find related bugs
 - Helps identify underlying cause/trigger of bug
 - Can use to get simpler path to problem
 - simpler usually means easier to fix
- Find most serious consequences of defect
 - Can you turn garbage output into a crash

Outline



- What is testing?
 - Goals and nature of testing
 - Place in quality assurance strategy
- How to test?
 - Kinds of testing
 - Important techniques
 - Effective testing practices
- When to test?
 - Software lifecycle and process
- When to stop?
 - Testing metrics
 - How to decide when you're done

Lifecycle Issues



Lifecycle Issues



- Requirements
 - Testing the requirements document
 - Making requirements testable
- Design
 - Testing the design
 - Designing for testability

Defect Tracking



- Provides a way of measuring quality
 - And progress towards quality
- Facilitates communication
 - Organized record and status of each defect
 - Ensures problems are not forgotten
- Provides accountability
 - Can identify and fix problems in process
 - Not enough detail in test reports
 - Not rapid enough response to bug reports
 - Should not be used for evaluation
- Your experiences?

Test Plan



- A test plan documents the overall strategy to be used in testing a system
 - Goals of testing
 - Quality targets, measurable if possible
 - What will be tested/what will not
 - Don't forget quality attributes!
 - Approach: techniques to be used
 - Schedule for testing (priority, etc.)
 - Organization (division of labor, etc.)
 - Anticipated cost
 - Criteria for completeness
 - Deliverables

Why Produce a Test Plan?



- May be a required product
 - Customer demands for maintenance
 - Part of all military contracts
 - Required for safety-critical certification
 - e.g. aircraft software
 - Someone else may implement the plan
- May benefit you
 - In this case, use as a tool: only document what provides value to you

Benefits of Test Plans



- Test quality
 - Improve coverage via list of features and quality attributes
 - Analysis of program (e.g. boundary values)
 - Avoid repetition and check completeness
- Communication
 - Get feedback on strategy
 - Agree on cost, quality with management
- Organization
 - Divide the work in a sensible way
 - Accountability for members of the team
 - Measure status

Strategy Statement

from Kaner, Bach Pettichord, Lessons Learned in Software Testing



- High level summary of testing strategy
- Examples:
 - We will release the product to friendly users after a brief internal review to find any truly glaring problems. The friendly users will put the product into service and tell us about any changes they'd like us to make.
 - We will define use cases in the form of sequences of user interactions with the product that represent, altogether, all the ways we expect normal people to use the product. We will augment that with stress testing and abnormal use testing (invalid data and error conditions). Our top priority is finding fundamental deviations from specified behavior, but we're also concerned with ways in which this program might violate user expectations. Reliability is a concern, but we haven't yet decided how best to evaluate that.
 - We will perform parallel exploratory testing and automated regression test development and execution. The exploratory testing will focus on validating basic functions (capability testing) to provide an early warning system for major functional failures. We will also be alert to opportunities for high-volume random testing.

Test Plan Development



- Likely to be incremental
 - You will get the strategy wrong the first time
 - As you discover more about the program, you will revise your strategy
- Focus efforts on risk and value
 - Most likely errors
 - Biggest impact errors
 - Hardest to fix errors

Test Plan Organization



- Lists and Tables
 - Dialogs/Reports/Screens
 - Input and Output Variables
 - Relationships between them
 - Which dialogs/reports/screens read/write them
 - Valid data ranges (and boundary values)
 - Features and functions
 - Use outlining to focus on detail
 - Error messages

Outline



- What is testing?
 - Goals and nature of testing
 - Place in quality assurance strategy
- How to test?
 - Kinds of testing
 - Important techniques
 - Effective testing practices
- When to test?
 - Software lifecycle and process
- When to stop?
 - Testing metrics
 - How to decide when you're done

When are you done testing?



When are you done testing?



- Coverage criterion
 - Must reach X% coverage
 - Legal requirement to have 100% coverage for avionics software
 - Distorts the software
- Can look at historical data
 - How many bugs are remaining, based on matching current project to past experience?
 - Key question: is the historical data applicable to a new project?

When are you done testing?



- Can use statistical models
 - Test on a realistic distribution of inputs, measure % of failed tests
 - Ship product when quality threshold is reached
 - Only as good as your characterization of the input
 - Usually, there's no good way to characterize this
 - Exception: stable systems for which you have empirical data (telephones)
 - Exception: good mathematical model (avionics)

Mutation Testing [slide adapted from Scherlis]



- Perturb code slightly in order to assess sensitivity
 - Focus on low-level design decisions
 - Change "<" to ">"
 - Change "0" to "1"
 - Change "<=" to "<"
 - Change "argv" to "argx"
 - Change "a.append(b)" to "b.append(a)"
 - Coverage criterion: % of mutants caught
 - Is it possible to catch them all?
 - Principle: % of mutants not found ~ % of errors not found
 - Is this really true?
 - Depends on how well mutants match real errors
 - Some evidence of similarity (e.g. off by one errors) but clearly imperfect

When are you done testing?



- Rule of thumb: when error detection rate drops
 - Implies diminishing returns for testing investment
- Most common
 - Run out of time or money
- Ultimately a judgment call
 - Resources available
 - Schedule pressures
 - Available estimates of quality

For More Information



- Kaner, Falk, Nguyen. Testing Computer Software (2nd Edition).
- One of my primary sources for this lecture

