

Testing

- Objectives of software testing
- Types of testing
- Testing strategy
- Reflections



CIS 422/522 Winter 2014

1

Testing Fundamentals

- Coding produces errors
 - Data show 30-85 errors are made per 1000 SLOC
- Testing: processes of executing the code to detect errors
- In practice, it is impossible to check for all possible errors by testing
- Even checking a useful subset is expensive
 - 40%-80% of development cost
 - Must be re-done when software changes
 - Potentially unbounded effort

CIS 422/522 Winter 2014

2

Testing Fundamentals (2)

- Reality: must settle for testing a subset of possible inputs
 - Even extensively tested software contains 0.5-3 errors per 1000 SLOC
 - Pesticide Paradox: *every method used to prevent or find bugs leaves a residue of subtler bugs against which those methods are ineffectual [Beizer]*
 - Always a tradeoff of cost vs. errors found
- Fundamental cost/benefit questions
 - Which subsets of possible test cases will find the most errors?
 - Which will find the most important errors?
 - How much testing is enough?

Ideal Testing Goal

- Goal: choose a sufficiently small but adequate set of test cases (input domain)
 - Small enough to economically run the complete set and re-run when software changes
 - “Adequate” much harder to define, generally means the test set is sufficient to show that the software is:
 - Acceptably close to required functional behavior
 - Contains no catastrophic faults
 - Reliable to an acceptable level (mean time to failure)
 - Within tolerance levels for qualities like performance, security, etc.

Testing Objectives

- Disagreement over best criteria for choosing the test set leads to two general approaches
- *Fault Detection*: testing intended to find as many faults as possible
- *Confidence Building*: testing intended to increase confidence that the software works as intended

Why continuing disagreement?

- Both approaches have significant weaknesses
- Fault Detection (bug hunt)
 - Tests according to coverage criteria
 - Equal chance, cost for finding arbitrary error
 - Implicitly assumes all bugs are equal, clearly not true in many cases
- Confidence Building (usage emulation)
 - Tests according to expected use
 - Higher chance of finding bugs that users will routinely encounter, misses others
 - Implicitly assumes that infrequent bugs are unimportant, also untrue in many cases

Methods by Adequacy Criteria

- Methods often classified by the criteria used to choose the test set
- Classification based on the source of information to derive test cases:
 - black-box testing (functional, specification-based)
 - white-box testing (structural, program-based)
- Classification based on the criterion to measure the adequacy of a set of test cases:
 - coverage-based testing
 - fault-based testing
 - error-based testing

White-Box Testing

- Also “clear box”
- Testing strategies based on knowledge of the code within a module
- Generally applies one or more forms of coverage criteria
 - Every non-commentary line of code is executed (statement coverage)
 - Every branch is taken (branch coverage)
 - Every block of code is executed (block coverage)
 - Every path is executed (path coverage)
 - Every defined variable is (correctly) used (define-use coverage)

Black-Box Testing

- Testing strategies using knowledge of interface specification, but not of implementation code
- For module tests:
 - Returned values conform to syntactic and semantic specifications for the interface
 - Inputs beyond parameter bounds, or that violate syntax or semantics, throw exceptions
 - Performance requirements are met (where defined)
- For integration and system tests
 - Sunny day, rainy day scenarios produce expected results
 - Can be based on use cases

Coverage Testing

- Looks at internal code structure (white-box)
- Test set adequacy defined by some form of coverage criteria
 - E.g., % of statements executed
- Three techniques:
 - control-flow coverage
 - data-flow coverage
 - coverage-based testing of requirements

Example: Control Flow Coverage

- Model program as flow graph
 - E.g., branches are nodes with multiple edges
 - An execution is one path through the graph
 - Generally very large number of possible paths
- Adequacy based on coverage of some aspect of the graph, in increasing order:
 - Node coverage: execute each statement
 - Branch coverage: execute each branch
 - Path coverage: execute every path
- % Coverage provides a test-set metric
- Many supporting tools

Example: Fault-based Testing

- Does not look at code structure (black-box)
- Looks for a test set with a high ability to detect faults
- Two techniques:
 - Fault seeding
 - Mutation testing

Fault Seeding

- Adequacy of test set judged by ability to find seeded errors
 - Seeds errors randomly into the code
 - Look at percentage of seeded errors found
 - Better test sets find more of the seed errors
- Infer that those sets will also find more latent errors
 - Look for high percentage of seeded to latent errors

Example: Operational Scenarios

- Focus on *confidence building* (rather than error-detection), also black-box
- Based on knowledge about how users (will) use the system
 - Inputs based on statistical sampling of actual inputs
 - Inputs based on estimates, use cases, user observation, focus groups, etc.
- Supports statistical inference about the likelihood of a failure in actual use (i.e., Cleanroom)
 - Usability requirements
 - Performance requirements
- Misses unlikely events
 - Low-frequency events tend not to be tested (edge cases, exceptions, unpredictable behavior)
 - Some low frequency events are critical (e.g., failure cases)

Experimental Results

- There is no uniformly best technique
- Different techniques tend to reveal different types of faults
- Multiple techniques reveal more faults (at a cost)
- Cost-effectiveness of run-time testing is low, particularly compared to inspections (most tests find no errors)
 - Design review: 8.44
 - Code review: 1.38
 - Testing: 0.17

Interpretation

- A combination of manual and automated techniques is most cost effective
 - People are better at detecting many kinds of errors than machines
 - Logic errors, misinterpretations, etc.
 - Machines are better at repetitive checks and minute details (comparing values)
- Testing works best in a supporting role (checking assumptions)
 - “Testing in quality” does not work (i.e., build then test to fix)
 - Activity of producing test cases and results double checks other artifacts
 - Is it well enough defined to write a good test case?
 - Are edge cases defined? Etc.
 - Gives feedback on assumptions and expectations: does the system do what we expect?

Best Approach

- Start early, test often
 - For every work product, we ask: *How can I find defects as early as possible?*
 - Create test plans and test cases as a way of checking the qualities of requirements, design, etc.
- Use a combination of methods
 - Inspections and reviews of every artifact
 - Testing at every stage possible
 - Manual
 - Module
 - System

Software Testing in Practice

- Most companies' new hires are testers
 - Regarded as less prestigious, lower skilled activity
- Most testing work is manual; help from tools is still limited
- In many cases, testing is not performed using systematic testing methods or techniques
- Often delayed, cut short by schedule pressure
- Sometimes there are "conflicts of interest" between testers and developers
 - Testing should be "destructive" as possible
 - Difficult attitude for developer
- Result is poor return for time/money spent

QA Planning

- Effective testing must be part of the overall plan
 - Fully supported by management (time, budget, skills)
 - Fully integrated into the development plan from the beginning
- Include use and evaluation of results
 - Process for addressing defects found
 - Measures of code quality
 - Measures of test quality and completeness
- Test results must provide feedback for improvement
 - Better QA process
 - Better coding practices, etc.
- Look at example plan

Quality is Cumulative

Requirements Analysis
Architectural Design
Detailed Design
Coding

- Are the requirements valid?
- Complete? Consistent? Implementable?
- Testable?
- Does the design satisfy requirements?
- Are all functional capabilities included?
- Are qualities addressed (performance, maintainability, usability, etc.?)
- Do the modules work together to implement all the functionality?
- Are likely changes encapsulated?
- Is every module well defined
- Implement the required functionality?
- Race conditions? Memory leaks? Buffer overflow?

Questions?