17-654/17-754
Analysis of Software Artifacts

Spring 2007

Testing (continued)

**All material © Jonathan Aldrich and William L Scherlis 2007
No part may be copied or used without written permission.**

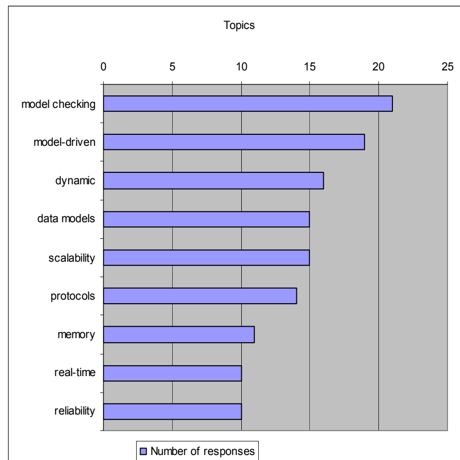
Jonathan Aldrich
Assistant Professor
Institute for Software Research

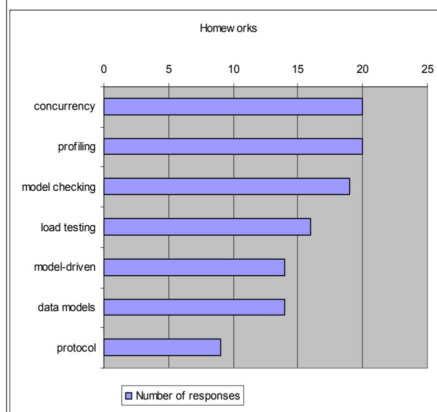
School of Computer Science
Carnegie Mellon University

jonathan.aldrich@cs.cmu.edu
+1 412 268 7278

Primary source: Kaner, Falk, Nguyen. Testing Computer Software (2nd Edition).

Topic Survey





Questions, comments, suggestions

- Is unit testing too dependent on the language?
  - jUnit has counterparts in other languages
    - especially nice in Java due to language features
  - Strategy will depend on language
    - Student: Java has built-in GC & concurrency, thus very different from C++
    - really, same could be said for ***all*** QA techniques
    - I hope to discuss language issues in more detail in a future lecture
- How do inspection and unit testing fit together?
  - Student: used inspection to find many bugs in HW
  - one of the good things about unit testing is you do inspect the code
  - we'll talk about these tradeoffs more in the next class, on inspection
- Comments, suggestions
  - Post the participation sheets
    - Done!
  - Show demos in class
    - Definitely—not every day but often
  - Backing up ideas with research is helpful
    - Good—I wish I had more of this to show! But a lot in SE is folklore.
  - Discussion w/ neighbors not always helpful – go straight to whole class
    - I agree, how useful this is depends on the topic. We'll adjust.
  - Balance time among topics
    - I want to respond to student needs/questions/opportunities dynamically, but we will try to find a balance

Testing – The Big Questions

1. **What is testing?**
   - And why do we test?
2. **What do we test?**
   - Levels of structure: unit, integration, system…
3. **How do we select a set of good tests?**
   - Value-driven testing
   - Functional (black-box) testing
   - Structural (white-box) testing
4. **How do we assess our test suites?**
   - Coverage, Mutation, Capture/Recapture…
5. **Practices for testability**
   - What are known best test practices?
   - How does testing integrate into lifecycle and metrics?
6. **What are the limits of testing?**
   - What are complementary approaches?
     - *Inspections*
     - *Static and dynamic analysis*

White Box Testing: Checkpoints

- Use "checkpoints" in code
  - Access to intermediate values
  - Enable checks *during* execution

Three approaches

- Logging
  - Create a log record of internal events
  - Tools to support
    - java.util.Logging
    - org.apache.log4j
  - Log records can be analyzed for patterns of events
    - Listener events
    - Protocol events
    - *Etc.*
- Assertions
  - Logical statements explicitly checked during test runs
  - (No side effects on program variables)
  - Check data integrity
    - Absence of null pointer
    - Array bounds
    - *Etc.*
- Breakpoints
  - Provide interactive access to intermediate state when a condition is raised

Benefits of White-Box

- Tool support can measure coverage
  - Helps to evaluate test suite (careful!)
  - 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
- Can find latent faults
  - Cannot trigger a failure in program, but can be found by a unit test

White Box: Limitations

- Is it possible to achieve 100% coverage?
- Can you think of a program that has a defect, even though it passes a test suite with 100% coverage?
- Exclusive focus on 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*

Black Box: Equivalence Class / Partition 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

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 an Equivalence Class?

- **Risk-based**
  - *Consider the cost of consequences*
    - Vs. frequency of occurrence
    - Focus test data around potential high-impact failures

Risk = (cost of consequence) * (probability of occurrence)

  - Challenge: How to model this set of high-consequence failures?
  - Selection heuristic – consider **boundary values**
    - Extreme or unique cases at or around "boundaries" with respect to preconditions or program decision points
      - *Examples*: zero-length inputs, very long inputs, null references, etc.
    - Will usually find errors that are present in any other member of the equivalence class, but may find off-by-one errors as well
  - Suited to **black box** and **white box**
  - **Input**: Information regarding fault/failure relationships
  - **Input**: Information regarding boundary cases
    - Requirements
    - Implementation

Robustness Testing

- *Test erroneous inputs and boundary cases*
  - Assess consequences of misuse or other failure to achieve preconditions
  - Bad use of API
  - Bad program input data
  - Bad files (e.g., corrupted) and bad communication connections
  - Buffer overflow (security exploit) is a robustness failure
    - Triggered by deliberate misuse of an interface.
- Test apparatus needs to be able to catch and recover from crashes and other hard errors
  - Sometimes multiple inputs need to be at/beyond boundaries
- The question of responsibility
  - Is there external assurance that preconditions will be respected?
  - *This is a design commitment that must be considered explicitly*

a[mid]

What if the array reference a is null?

Equivalence, Boundary and Robustness Example

- Program Specification
  - Given numbers a, b, and c, return the roots of the quadratic polynomial $ax^2 + bx + c$. Recall that the roots of a quadratic equation are given by: $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$
- Equivalence classes?
- Boundary test cases?
- Robustness test cases?

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?

Protocol Testing

- **Object protocols**
  - Develop test cases that involve representative sequence of operations on objects
    - Example: Dictionary structure
      - Create, AddEntry*, Lookup, ModifyEntry*, DeleteEntry, Lookup, Destroy
    - Example: IO Stream
      - Open, Read, Read, Close, Read, Open, Write, Read, Close, Close
  - Test concurrent access from multiple threads
    - Example: FIFO queue for events, logging, etc.

Create	Put		Put		Get	Get			
		Put		Get	Get		Put	Put	Get

- Approach
  - Develop representative sequences – based on use cases, scenarios, profiles
  - Randomly generate call sequences
    - Example: Account
      - Open, Deposit, Withdraw, Withdraw, Deposit, Query, Withdraw, Close
  - Coverage: Conceptual states
- Also useful for protocol interactions within distributed designs

Testing example

- Test **preparation**
  - Client scaffold
  - Failure recovery: exceptions
- Test case **selection**
  - *Expected* cases
    - key found
    - key not found
  - *Extreme* cases
    - empty array
    - singleton array
    - large array
  - *"Sub-unit" testing*
    - ordering relation over domain (…)
  - *Non-functional* testing
    - Performance measurement
      - Expectation: algorithmic analysis
      - Broken code: can yield a linear-time implementation vs. log-time
        - E.g., 1m elements: 20 steps vs. 1,000,000 steps
- **Coverage** analysis
  - Statement, branch, path coverage
  - Data coverage
- **Static analysis** and **inspection**
  - Initialization; array bounds; arithmetic exceptions; coding style

```
public static int binsrch (int[] a, int key) {
    int low  = 0;
    int high = a.length - 1;
    while (true) {
        if ( low > high ) return -(low+1);
        int mid = (low+high) / 2;
        if      ( a[mid] < key )  low  = mid + 1;
        else if ( a[mid] > key )  high = mid - 1;
        else    return mid;
    }
}
```

Discussion: What is a Bug?

- What about a misleading comment?
- What if a piece of code seems wrong, but can't actually lead to a failure when you run the program?
- Is a bug OK if it is there to enhance performance?

17-654 Spring 2007 –Aldrich © 2007

Discussion: What is a Bug?

- A flaw in a software artifact that could lead to a program's failure to meet its specification (or satisfy its users)
  - In the fault/error/failure terminology, a bug is a fault
  - Artifact: code, test, design, specification, …
  - Could lead to a failure: consider software evolution
  - Specification/users: focus on *intent* to determine if it's a bug
- What is the effect?
  - It's a bug if it leads to failure (violation of spec)
  - It's a bug if it leads to an error condition (violation of internal invariant)
- Bug or feature?
  - What is the *intended* behavior?
    - e.g. A program gives a result that is close to but not exactly to the mathematical answer. If the specification allows a margin for error, this is OK. That might be a rational choice if other quality attributes are more important, e.g. performance.
- Code is wrong but case can't be executed
  - What is the intended path to software evolution?
    - If that path might be executed in the future, this is a bug
- Comment bugs
  - Could lead to defects being introduced as code evolves
  - Confusingly written code is a bug for the same reason
- Specification bugs
  - Omission: does not define which behavior is correct
  - Validation: does not capture what the user(s) need

17-654 Spring 2007 –Aldrich © 2007

Unit Testing and Coverage Takeaways

- Testing is direct execution of code on test data in a controlled environment
  - Testing *can* help find bugs, assess quality, clarify specs, learn about programs, and verify contracts
  - Testing *cannot* verify correctness
- Unit testing has multiple benefits
  - Clarifies specification
  - Isolates defects
  - Finds errors as you write code
  - Avoids rework
- Coverage critieria useful for structuring tests
  - Whitebox – coverage of program constructs
    - Lines, branches, methods, paths, etc.
    - Useful to tell you where you are missing tests
    - Not sufficient to guarantee adequacy
  - Blackbox – coverage of specification
    - Partition testing, boundary testing, robustness testing
    - Often better guide for writing tests

Testing – The Big Questions

1. **What is testing?**
   - And why do we test?
2. **What do we test?**
   - Levels of structure: unit, integration, system…
3. **How do we select a set of good tests?**
   - Value-driven testing
   - Functional (black-box) testing
   - Structural (white-box) testing
4. **How do we know when we're done?**
   - Coverage, Mutation, Capture/Recapture…
5. **Practices for testability**
   - What are known best test practices?
   - How does testing integrate into lifecycle and metrics?
6. **What are the limits of testing?**
   - What are complementary approaches?
     - *Inspections*
     - *Static and dynamic analysis*

When are you done testing?

- Coverage criterion
  - Must reach X% coverage
    - Legal requirement to have 100% coverage for avionics software
    - Drawback: focus on 100% coverage can distort the software so as to avoid any unreachable code
- 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?
- 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)
  - Caveat: random generation from a known distribution is good for estimating quality, but generally not good at finding errors
    - Errors are more likely to be found on uncommon paths that random testing is unlikely to find
- Rule of thumb: when error detection rate drops
  - Implies diminishing returns for testing investment

When are you done testing?

- **Mutation testing**
  - *Perturb code slightly in order to assess sensitivity*
  - Focus on low-level design decisions
    - Examples:
      - Change "<" to ">"
      - Change "0" to "1"
      - Change "≤" to "<"
      - Change "argv" to "argx"
      - Change "a.append(b)" to "b.append(a)"
- Assess effectiveness of test suite
  - How many seeded defects are found?
    - coverage metric
  - 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 inspecting?

- Capture/Recapture assessment
  - Most applicable for assessing inspections
  - Measure overlap in defects found by different inspectors
  - Use overlap to estimate number of defects not found
- Example
  - Inspector A finds n1=10 defects
  - Inspector B finds n2=8 defects
  - m = 5 defects found by both A and B
  - N is the (unknown) number of defects in the software
- Lincoln-Petersen analysis [source: Wikipedia]
  - Consider just the 10 (total) defects found by A
  - Inspector B found 5 of these 10 defects
  - Therefore the probability that inspector B finds a given defect is 5/10 or 50%
  - So, inspector B should have found 50% of the N defects in the software, so

$$N = n1 * n2 / m = 10 * 8 / 5 = 20 \text{ defects}$$

- Assumptions
  - All defects are equally easy to find
  - All inspectors are equally effective at finding defects
  - Are these realistic?

When are you done testing?

- Most common
  - Run out of time or money
- Ultimately a judgment call
  - Resources available
  - Schedule pressures
  - Available estimates of quality

Testing – The Big Questions

1. **What is testing?**
   - And why do we test?
2. **What do we test?**
   - Levels of structure: unit, integration, system…
3. **How do we select a set of good tests?**
   - Value-driven testing
   - Functional (black-box) testing
   - Structural (white-box) testing
4. **How do we know when we're done?**
   - Coverage, Mutation, Capture/Recapture…
5. **Practices for testability**
   - What are known best test practices?
   - How does testing integrate into lifecycle and metrics?
6. **What are the limits of testing?**
   - What are complementary approaches?
     - *Inspections*
     - *Static and dynamic analysis*

17-654 Spring 2007 –Aldrich © 2007

5a. Practices for testability

1. Document interfaces
   - Write down explicit "rules of the road" at interfaces, APIs, etc

- Design by contract
  - Specify a contract between service **client** and its **implementation**
    - System works if both parties fulfill their contract
    - Use pre- and post-conditions, etc
- Testing
  - Verify pre- and post-conditions during execution
    - Important Limitation
      - Not all logical formulas can be evaluated directly (forall x in S…)
  - Assign responsibility based on contract expectations
  - Executions become a set of unit tests

4.4.2 delete_binding

The delete_binding API call causes one or more instances of bindingTemplate d
registry.

4.4.2.1 Syntax:

```
<delete_binding generic="2.0" xmlns="urn:uddi-org:api_v2" >
????????? <authInfo/>
????????? <bindingKey/> [<bindingKey/> ?]
</delete_binding>
```

4.4.2.2 Arguments:

? ***authInfo***: this required argument is an element that contains an authenti
tokens are obtained using the get_authToken API call.

? ***bindingKey***: one or more *uuid_key* values that represent specific instanc

4.4.2.3 Returns:

Upon successful completion, a dispositionReport is returned with a single succe
bindingTemplates that are deleted as a result of this call, such as those referenc
hostingRedirector elements) are not affected.

4.4.2.4 Caveats:

If any error occurs in processing this API call, a dispositionReport structure will
Fault.? The following error number information will be relevant:

? **E_invalidKeyPassed**: signifies that one of the *uuid_key* values passed
bindingKey values.? No partial results will be returned ? if any bindingKey
the message contained multiple instances of a uuid_key value, this error
caused the problem will be clearly indicated in the error text.

? **E_authTokenExpired**: signifies that the authentication token value pass

17-654 Spring 2007 –Aldrich © 2007

5a. Integration/System Testing

2. Do incremental 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

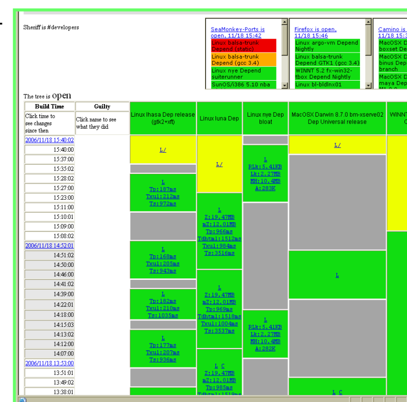
- Test interactions between modules
  - Ultimately leads to end-to-end system test

- Used focused tests
  - Set up subsystem for test
  - Test specific subsystem- or system-level features
    - no "random input" sequence
  - Verify expected output

5a. Frequent (Nightly) Builds

3. Build 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
   - Use test automation
     - Upfront cost, amortized benefit
     - Not all tests are easily automated – manually code the others

- Run simplified "smoke test" on build
  - Tests basic functionality and stability
  - Often: run by programmers before check-in
  - Provides rough guidance prior to full integration testing



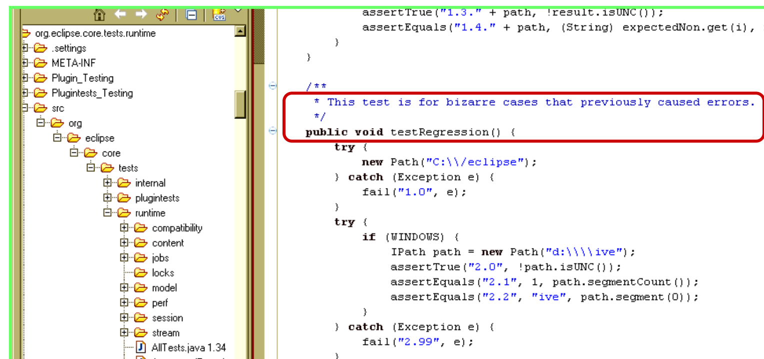
Practices –Regressions

4. Use regression tests

- Regression tests: run every time the system changes

• Goal: catch new bugs introduced by code changes

- Check to ensure fixed bugs stay fixed
  - New bug fixes often introduce new issues/bugs
- Incrementally add tests for new functionality



Practices – Acceptance, Release, Integrity Tests

5. Acceptance tests (by customer)

- Tests used by customer to evaluate quality of a system
- Typically subject to up-front negotiation

6. Release Test (by provider, vendor)

- Test release CD
  - Before manufacturing!
- Includes configuration tests, virus scan, etc
- Carry out entire install-and-run use case

7. Integrity Test (by vendor or third party)

- Independent evaluation before release
- Validate quality-related claims
- Anticipate product reviews, consumer complaints
- Not really focused on bug-finding

Practices: Reporting Defects

8. Develop good defect reporting practices

- Reproducible defects
  - Easier to find and fix
  - Easier to validate
    - Built-in regression test
  - Increased confidence
- Simple and general
  - More value doing the fix
  - Helps root-cause analysis
- Non-antagonistic
  - State the problem
  - Don't blame

eclipse Eclipse bugs
Bugzilla Bug 141261 crash - Shell create, RepositionWindow() - Unexpected Eclipse crash RC3 (JavaNativeCrash)
Last modified: 2006-11-14 17:46:58

Practices: Social Issues

9. Respect social issues of testing

- There are differences between developer and tester culture
- Acknowledge that testers often deliver bad news
- Avoid using defects in performance evaluations
  - Is the defect real?
  - Bad will within team
- Work hard to detect defects before integration testing
  - Easier to narrow scope and responsibility
  - Less adversarial
- Issues vs. defects

View Bug Activity | Format For Printing | Clone This Bug

Description: [reply] Opened: 2005-07-25 07:03

I didn't even know that there was an undo feature inside the GUI editor, but today I accidently pressed CTRL-Z instead of CTRL-S and the undo started... crashed.

Try adding some extension in the extensions page and then press CTRL-Z.

This is actually two bugs imho;

1. The details is very very poor so I really don't know what happened. A stacktrace would be great for debugging.

2. The undo obviously does not work correctly.

here is a screenshot of the crash:
http://mnemo.minimum.se/eclipse_crashes/eclipse_undo_crash.png

I don't have time for extensive reprod testing atm, maybe someone else can assist with this and see if they can get the plugin.xml editor to crash using weird combinations of editing and CTRL-Z undoing.

Practices: Root cause analysis

10. How can defect analysis help prevent later defects?

- Identify the “root causes” of frequent defect types, locations
  - Requirements and specifications?
  - Architecture? Design? Coding style? Inspection?
- Try to find all the 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 root cause of the defect
  - Can use to get simpler path to problem
    - This can mean easier to fix
- Identify the most serious consequences of a defect

Testing – The Big Questions

1. **What is testing?**
   - And why do we test?
2. **What do we test?**
   - Levels of structure: unit, integration, system…
3. **How do we select a set of good tests?**
   - Value-driven testing
   - Functional (black-box) testing
   - Structural (white-box) testing
4. **How do we know when we’re done?**
   - Coverage, Mutation, Capture/Recapture…
5. **Practices for testability**
   - What are known best test practices?
   - How does testing integrate into lifecycle and metrics?
6. **What are the limits of testing?**
   - What are complementary approaches?
     - *Inspections*
     - *Static and dynamic analysis*

5b. Testing and Lifecycle Issues

1. Testing issues should be addressed at every lifecycle phase

- Initial negotiation
  - Acceptance evaluation: evidence and evaluation
  - Extent and nature of specifications
- Requirements
  - Opportunities for early validation
  - Opportunities for specification-level testing and analysis
  - Which requirements are testable: functional and non-functional
- Design
  - Design inspection and analysis
  - Designing for testability
    - Interface definitions to facilitate unit testing
- Follow both top-down and bottom-up unit testing approaches
  - Top-down testing
    - Test full system with stubs (for undeveloped code).
    - Tests design (structural architecture), when it exists.
  - Bottom-up testing
    - Units → Integrated modules → system

Lifecycle issues

2. Favor unit testing over integration and system testing

- Unit tests find defects earlier
  - Earlier means less cost and less risk
  - During design, make API specifications specific
    - Missing or inconsistent interface (API) specifications
    - Missing representation invariants for key data structures
    - What are the unstated assumptions?
      - Null refs ok?
      - Pass out this exception ok?
      - Integrity check responsibility?
      - Thread creation ok?
- Over-reliance on system testing can be risky
  - Possibility for finger pointing within the team
  - Difficulty of mapping issues back to responsible developers
  - Root cause analysis becomes blame analysis

Test Plan

3. Create a QA plan document

- Which quality techniques are used and for what purposes

- Overall system strategy
  - Goals of testing
    - Quality targets
    - Measurements and measurement goals
  - What will be tested/what will not
    - Don't forget quality attributes!
  - Schedule and priorities for testing
    - Based on hazards, costs, risks, etc.
  - Organization and roles: division of labor and expertise
  - Criteria for completeness and deliverables

1 Scope
  1.1 System Overview . . . .
2 Reference Documents
3 Software Test Environment
4 Test Identification
  4.1 General Information . . . .
    4.1.1 Test Level . . . .
    4.1.2 Test Classes . . . .
  4.2 Planned Tests . . . .
    4.2.1 Test 1 – Linear Operators . . . .
    4.2.2 Test 2 – Convergence of Multifluid Project
    4.2.3 Test 3 – Fixed-boundary diffusion solver
    4.2.4 Test 4 – Upwind advection . . . . . .
    4.2.5 Test 5 – Fixed-boundary projection test .
    4.2.6 Test 6 – Surface Tension Test . . . . . .
    4.2.7 Test 7 – Multifluid system test . . . . .
    4.2.8 Test 8 – Multifluid AMR test . . . . . .
    4.2.9 Test 9 – Multifluid system regression test
5 Test Schedules
6 Bug Tracking
7 Requirements Traceability

- Make decisions regarding when to unit test
  - There are differing views
    - **CleanRoom**: Defer testing. Use separate test team
    - ✓ **Agile**: As early as possible, even before code, integrate into team

Test Strategy Statement

- 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 … 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 will also use exploratory testing to identify ways in which this program might violate user expectations.

  - 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 pursue high-volume random testing where possible in the code.

[adapted from Kaner, Bach, Pettichord, Lessons Learned in Software Testing]

Why Produce a Test Plan?

4. Ensure the test plan addresses the needs of stakeholders

- Customer: may be a required product
  - Customer requirements for operations and support
  - Examples
    - Government systems integration
    - Safety-critical certification: avionics, health devices, etc.
- A separate test organization may implement part of the plan
  - "IV&V" – Independent verification and validation
- May benefit development team
  - Set priorities
    - Use planning process to identify areas of hazard, risk, cost
- Additional benefits – the plan is a team product
  - 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
    - Division of labor
    - Measurement of progress

Defect Tracking

5. Track defects and issues

- Issue: Bug, feature request, or query
  - May not know which of these until analysis is done, so track in the same database (Issuezilla)
- Provides a basis for measurement
  - Defects reported: which lifecycle phase
  - Defects repaired: time lag, difficulty
  - Defect categorization
  - Root cause analysis (more difficult!)
- Provides a basis for division of effort
  - Track diagnosis and repair
  - Assign roles, track team involvement
- Facilitates communication
  - Organized record for each issue
  - Ensures problems are not forgotten
- Provides some 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 HR evaluation

Testing – The Big Questions

1. **What is testing?**
   - And why do we test?
2. **What do we test?**
   - Levels of structure: unit, integration, system…
3. **How do we select a set of good tests?**
   - Value-driven testing
   - Functional (black-box) testing
   - Structural (white-box) testing
4. **How do we know when we're done?**
   - Coverage, Mutation, Capture/Recapture…
5. **Practices for testability**
   - What are known best test practices?
   - How does testing integrate into lifecycle and metrics?
6. **What are the limits of testing?**
   - What are complementary approaches?
     - *Inspections*
     - *Static and dynamic analysis*

6. What are the limits of testing?

- **What we can test**
  - Attributes that can be directly evaluated externally
    - *Examples*
      - **Functional** properties: result values, GUI manifestations, etc.
  - Attributes relating to resource use
    - Many well-distributed **performance** properties
    - Storage use
- **What is difficult to test?**
  - Attributes that **cannot easily be measured externally**
    - Is a design evolvable? — Design Structure Matrices
    - Is a design secure? — Secure Development Lifecycle
    - Is a design technically sound? — Alloy; see also Models
    - Does the code conform to a design? — ArchJava; Reflexion models; Framework usage
    - Where are the performance bottlenecks? — Performance analysis
    - Does the design meet the user's needs? — Usability analysis
  - Attributes for which **tests are nondeterministic**
    - Real time constraints — Rate monotonic scheduling
    - Race conditions — Analysis of locking
  - Attributes relating to the **absence of a property**
    - Absence of security exploits — Microsoft's Standard Annotation Language
    - Absence of memory leaks — Cyclone, Purify
    - Absence of functional errors — Hoare Logic
    - Absence of non-termination — Termination analysis

Assurance beyond Testing and Inspection

- Design analysis: check correctness early
  - Design Structure Matrices – evolvability analysis
  - Security Development Lifecycle – architectural analysis for security
  - Alloy – systematically exploring a model of a design
- Static analysis: provable correctness
  - Reflexion models, ArchJava – conformance to design
  - Fluid – concurrency analysis for race conditions
  - Metal, Fugue – API usage analysis
  - Type systems – eliminate mechanical errors
  - Standard Annotation Language – eliminate buffer overflows
  - Cyclone – memory usage
- Dynamic analysis: run time properties
  - Performance analysis
  - Purify – memory usage
  - Eraser – concurrency analysis for race conditions
  - Test generation and selection – lower cost, extend range of testing
- Manual analysis: human verification
  - Hoare Logic – verification of functional correctness
  - Real-time scheduling