Announcements

- Reminder: Assignment 3 due Tuesday at 5pm!
  - Different day/time than usual
- Questions on Assignment 3?

Testing Retrospective

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

Jonathan Aldrich
Assistant Professor
Institute for Software Research

School of Computer Science
Carnegie Mellon University

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

Nice Catch!

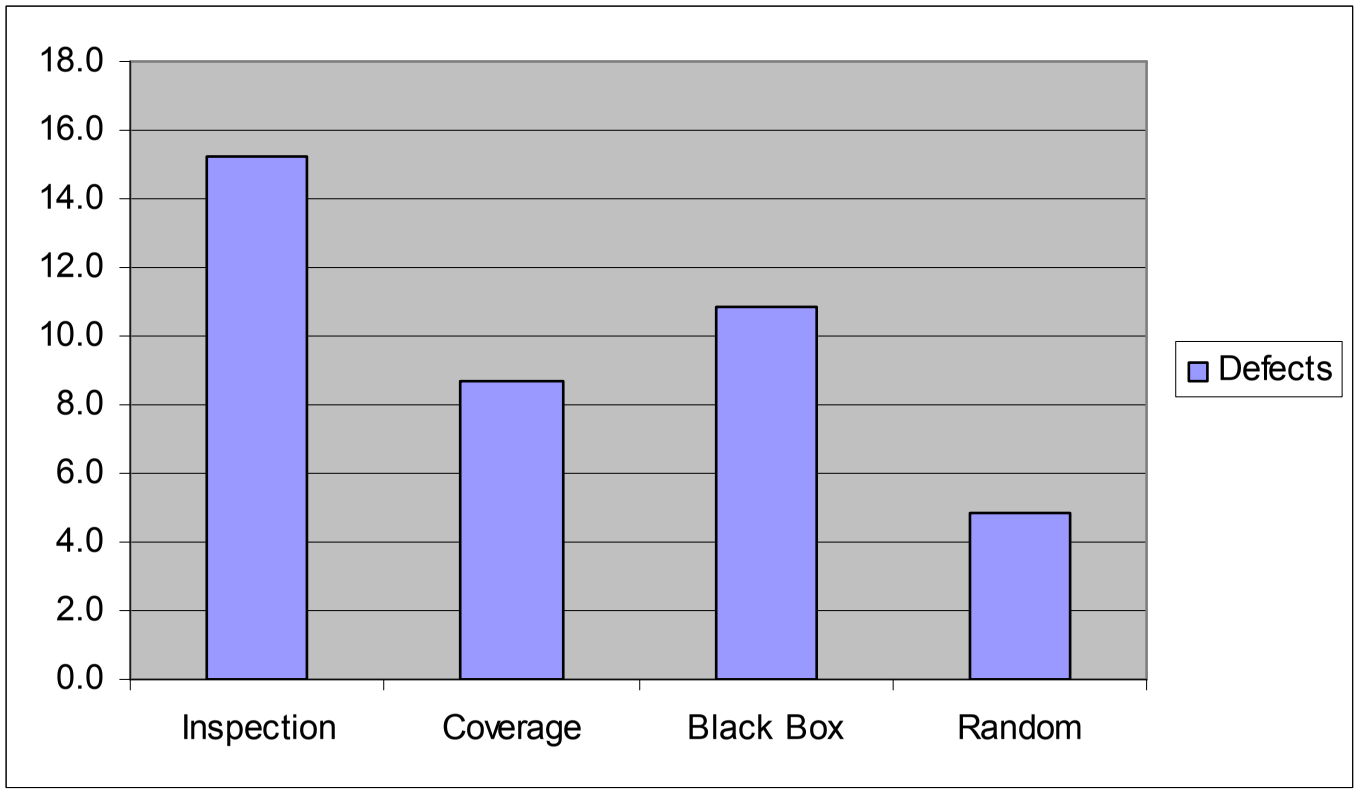

NFL SAFE
MANAGEMENT
SECURITY
10
SUPER BOWL XLIII
PITTSBURGH 27, ARIZONA 23
Steelers wide receiver Santonio Holmes (10) catches the game-winning touchdown pass in front of Cardinals safety Aaron Francisco with 35 seconds left in the game. Holmes finished with nine catches for 131 yards and was named Super Bowl MVP.
MATT CASHORE/US PRESSWIRE
ZOOM

Homework 2 – Testing Retrospective

- Summary statistics – with lots of caveats
  - Different implementations
  - Different testing strategies
  - Different effort levels
  - Small code base
  - Well-defined, limited domain (game playing rules)
  - etc.
- Despite the caveats, results appear robust
  - Fairly consistent across groups
  - Fit data from the research literature

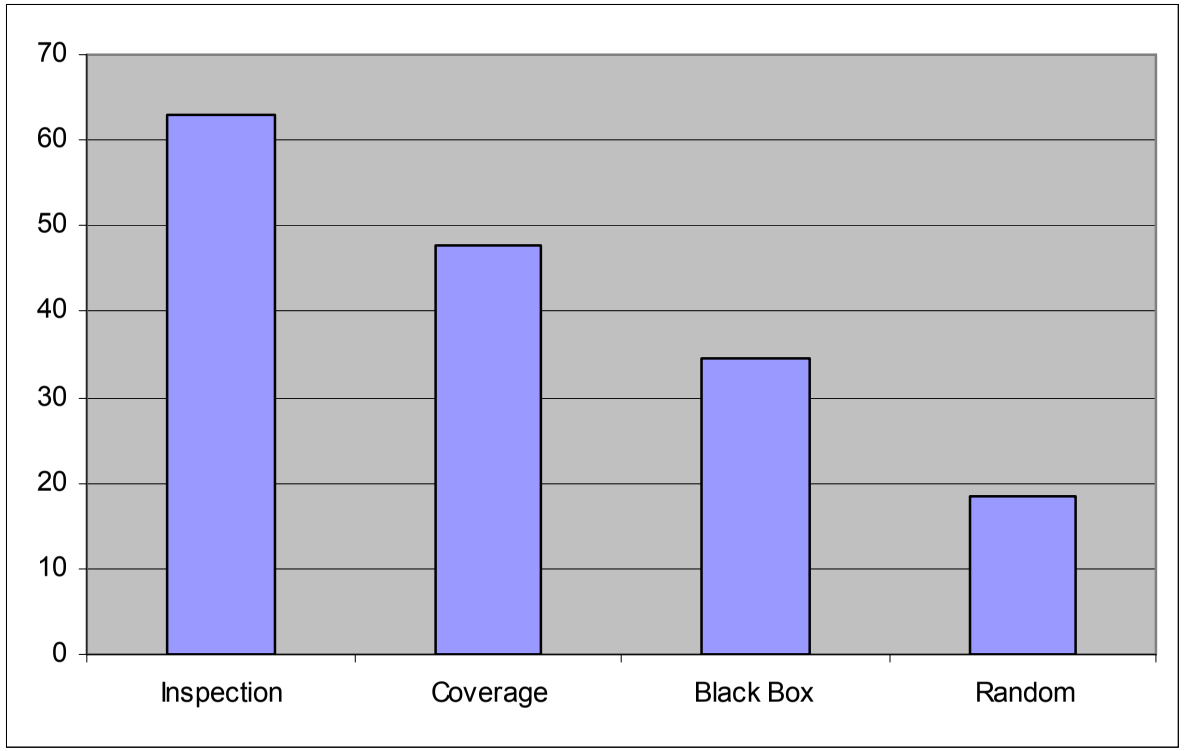
Which Technique Found the Most Defects?

Average # of Output-Affecting Defects Found



Which Technique is Most Cost-Effective?

Average # Defects Found / Average % Time Spent



Broad Conclusions

- Inspection is effective
  - Most cost-effective
    - even if you only consider output-affecting defects
    - even more so if you add in other defects
  - Most total defects
- Coverage is cost-effective, but is not comprehensive
  - Second only to inspection in cost-effectiveness
  - Found fewer defects than inspection or black-box
- Black-box testing is fairly comprehensive
  - Most defects found, after inspection
- Random testing is not very good
  - Worst in both categories
  - However, was competitive with other techniques for 2 groups
    - So perhaps it depends on how you do it!
- Do you think these will hold up generally?

Observations - Inspection

- Can find defects that doen't affect output
- Identifies root cause immediately
- Seems like a generally good idea

Observations - Coverage

- Good for finding dead code
- Seemed easy for some teams, hard for others
  - Especially when testing code with complex control structure
  - Lesson: keep your control flow as simple as possible!
- Covering every line finds many bugs—but definitely not all!
  - Cheap way to find additional defects if your line coverage is low
  - Best to complement with other techniques

Observations – Black Box

- Hard to categorize errors
- Quite a few robustness errors found
- Not as cost-effective as inspection or code coverage
  - but finds more defects than code coverage
  - but can use in regression, unlike inspection

Observations – Random Testing

- Some strategies that were used
  - Randomly generate moves & compare to oracle
    - Ideal precision, but often infeasible to implement
  - Randomly generate moves & apply pre/post-condition testing
    - Most practical approach
  - Randomly generate moves, print, and hand-inspect output
    - very inefficient! random testing only works if you can evaluate the output automatically, because the tests overlap so much
  - 2 groups – use an outside testing tool
    - Typically based on pre/post specification
    - Seemed quite effective, in both cases! 6 & 7 defects found

- Some notes from different teams
  - Easy to get stymied
    - null pointer exceptions kept from testing
    - error in move class kept from testing

Random Testing Tool Experiences?

Assignment 1 Inspection

- Lots of defects in the specification
- Some in the test code I gave you!

- How did inspecting the spec compare to inspecting code?

Other homework comments, questions?

Testing – The Big Questions

1. **What is testing?**
   - And why do we test?
2. **To what standard do we test?**
   - Specification of behavior and quality attributes
3. **How do we select a set of good tests?**
   - Functional (black-box) testing
   - Structural (white-box) testing
4. **How do we assess our test suites?**
   - Coverage, Mutation, Capture/Recapture…
5. **What are effective testing practices?**
   - Levels of structure: unit, integration, system…
   - Design for testing
   - Effective testing 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*

Discussion: 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
    - 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 (per unit of effort) 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

$$\mathbf{N = n1 * n2 / m} = 10 * 8 / 5 = 16 \text{ defects}$$

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

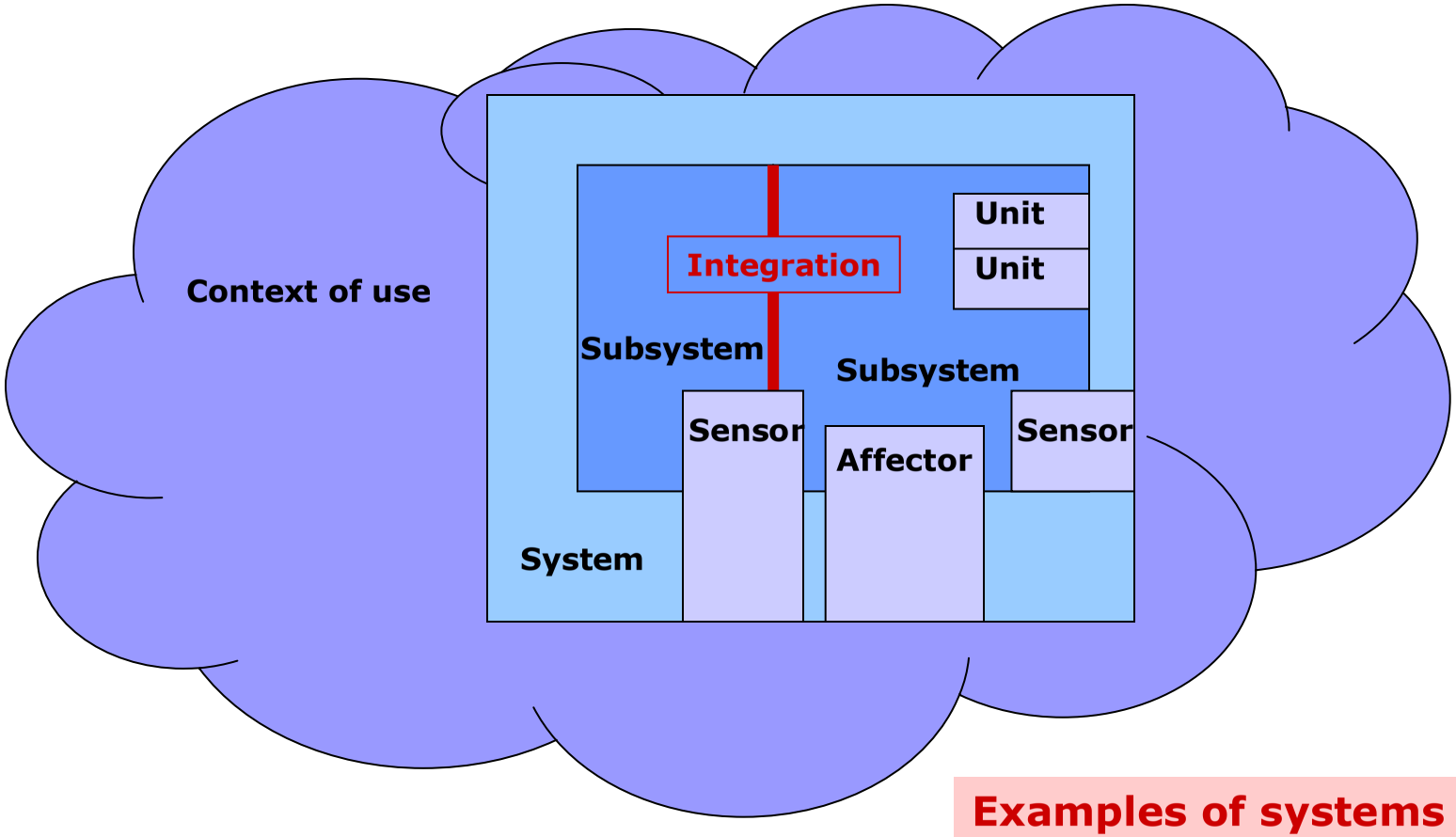
When are you done testing?

- 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. **To what standard do we test?**
   - Specification of behavior and quality attributes
3. **How do we select a set of good tests?**
   - Functional (black-box) testing
   - Structural (white-box) testing
4. **How do we assess our test suites?**
   - Coverage, Mutation, Capture/Recapture…
5. **What are effective testing practices?**
   - Levels of structure: unit, integration, system…
   - Design for testing
   - Effective testing 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*

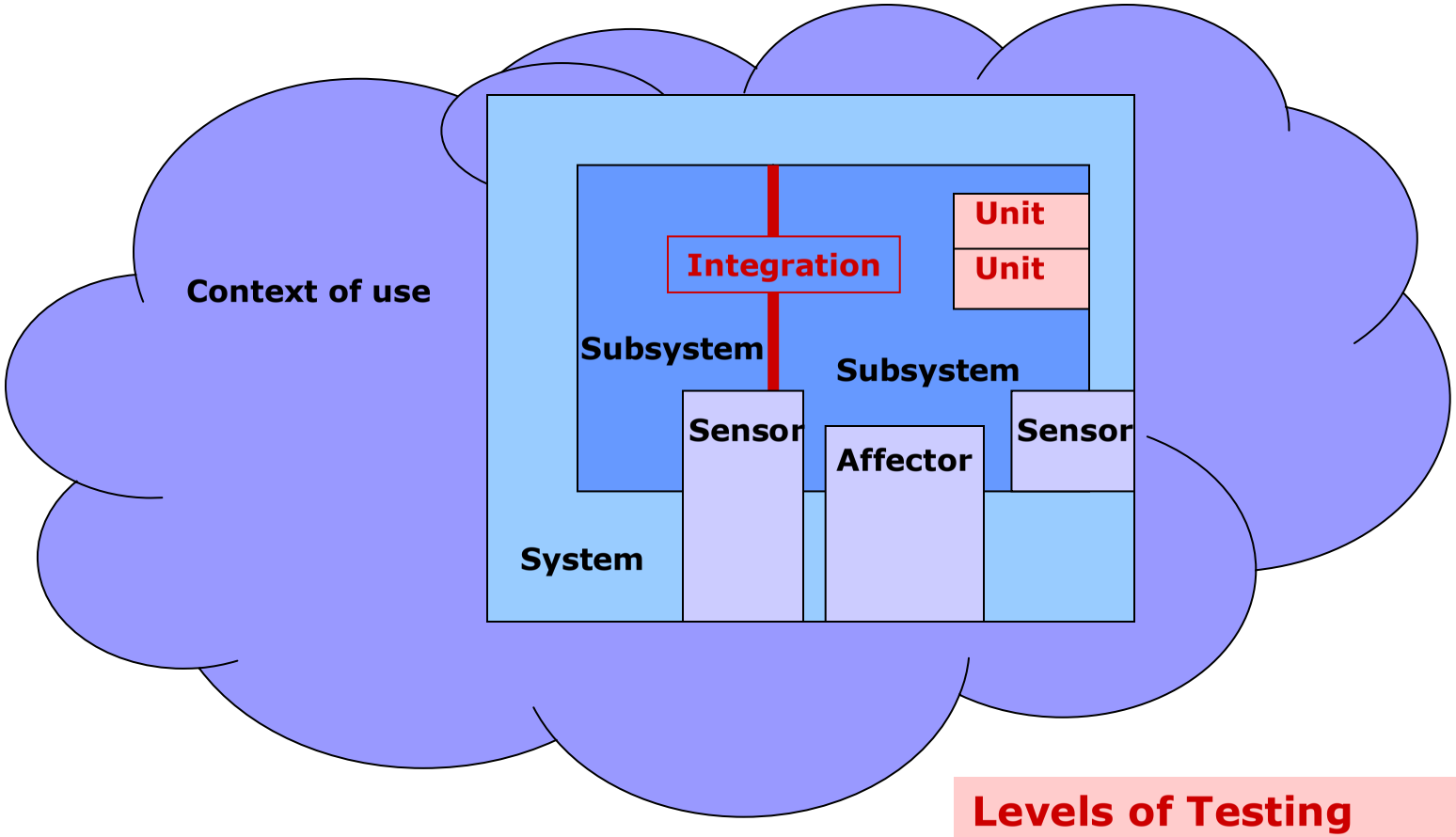
5. What do we test – the Focus of Concern



Examples of systems in context

- Mars rover
- Cell phone
- Clothes washing machine
- Point of sale system
- Telecom switch
- Software development tool

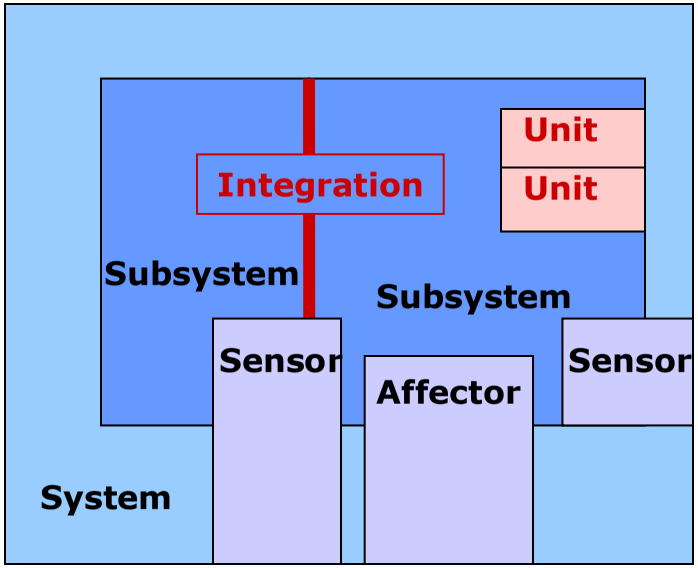
The Focus of Concern



Levels of Testing

- User testing, field testing

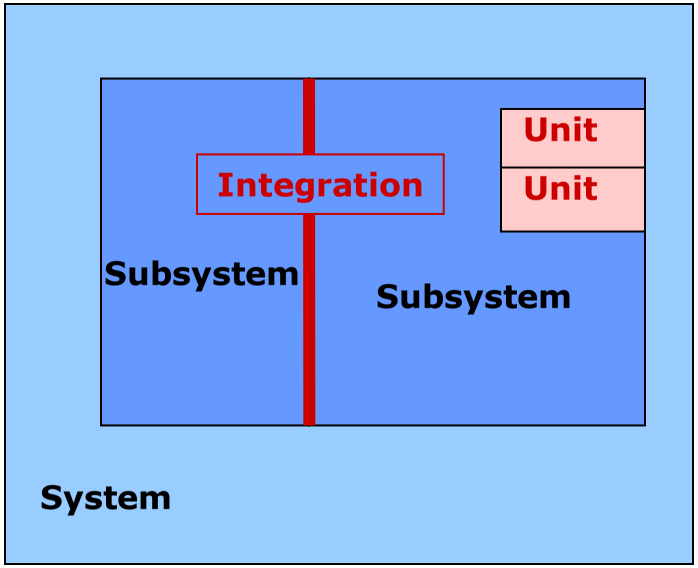
The Focus of Concern



Levels of Testing

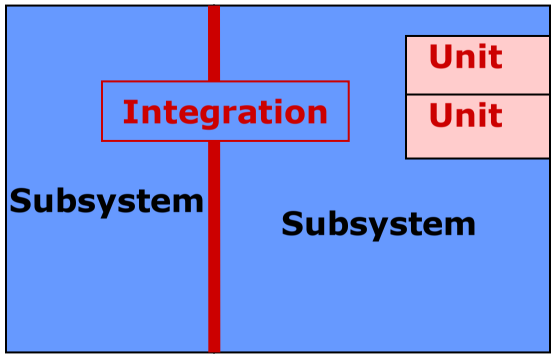
- User testing, field testing
- System testing

The Focus of Concern



Levels of Testing
- User testing, field testing
- System testing
  - With or without hardware

The Focus of Concern



Levels of Testing
- User testing, field testing
- System testing
  - With or without hardware
- Integration testing

The Focus of Concern

Unit

Levels of Testing
- User testing, field testing
- System testing
  - With or without hardware
- Integration testing
- Unit testing

Unit Tests

- Unit tests are whitebox tests written by developers, and designed to verify small units of program functionality.
- Key Metaphor: I.C. Testing
  - Integrated Circuits are tested individually for functionality before the whole circuit is tested.
- Definitions
  - Whitebox – Unit tests are written with full knowledge of implementation details.
  - Developers – Unit tests are written by you, the developer, concurrently with implementation.
  - Small Units – Unit tests should isolate one piece of software at a time.
    - Individual methods and classes
  - Verify – Make sure you built 'the software right.' Testing against the contract.
    - Contrast this with validation

[source: Nels Beckman]

Test-Driven Development

- Write the tests before the code
  - Helps you think about corner cases when writing
  - Helps you think about interface design
- Write code only when an automated test fails
- If you find a bug through other means, first write a test that fails, then fix the bug
  - Bug won't resurface later
- Run tests as often as possible, ideally every time the code is changed
  - Having comprehensive unit tests allows you to refactor code with confidence
  - Without unit tests, code is fragile – changes might break clients!

[source: Donna Malayeri]

Potential Benefits of Unit Tests

- Helps localize errors
  - Failure indicates problem in the unit under test
- Find errors early
  - Unit tests are written during development, usually by developer
    - QA still does functional, acceptance, user testing, etc.
  - More expensive to fix defects found later by another team
    - Must isolate bugs to their source
    - Must re-learn old code to debug it
    - Must often redesign code that was fundamentally broken
- Avoid unnecessary functionality
  - Write test first, only write enough code to get it working
- Help document specification
  - Tests are an unambiguous (though perhaps low-level and incomplete) specification of behavior
- Improve code quality code
  - Helps developer deliver working code

[source: Nels Beckman]

Testing Harnesses

- Testing harnesses are tools that help manage and run your unit tests.

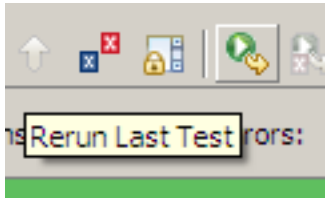
Help achieve three properties of good unit tests:

- Automatic
  - Tests should be easy to run and check for correct completion. This allows developers to quickly confirm their code is working after a change.
- Repeatable
  - Any developer can run the tests and they will work right away.
- Independent
  - Tests can be run in any order and they will still work.

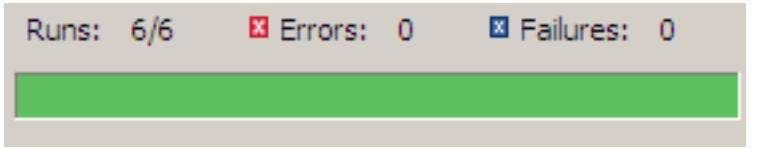
[source: Nels Beckman]

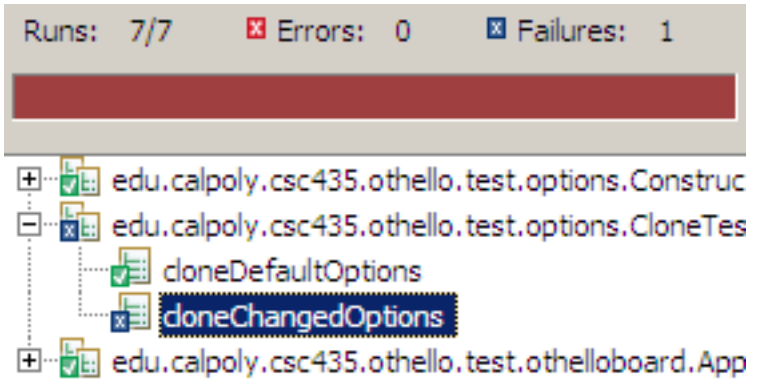
JUnit: A Java Unit Testing Harness

- Features
  - One click runs all tests
  - Visual confirmation of success or failure.
  - Source of failure is immediately obvious.



- JUnit framework interface
  - `@Test` annotation marks a test for the harness
  - `org.junit.Assert` contains functions to check results.





[source: Nels Beckman]

A JUnit Test Case

```
public class SampleTest  {
      private List<String> emptyList;

      @Before
      public void setUp() {
          emptyList = new ArrayList<String>();
      }

     @After
      public void tearDown() {
          emptyList = null;
      }

     @Test
      public void testEmptyList() {
          assertEquals("Empty list should have 0 elements",
                       0, emptyList.size());
      }
}
```

Helpful JUnit Assert Statements

- `assertTrue(boolean condition)`
- `assertFalse(boolean condition)`
  - Assert some condition is true (or false)

- `assertEquals(Object expected, Object actual)`
  - Check that some value is equal to another

- `assertEquals(float expected, float actual, float delta)`
  - Used for so that floating point equality is unnecessary.

- `assertSame(Object expected, Object actual)`
  - Tests for two objects are the same reference (identical) in memory.

- `assertNull(java.lang.Object object)`
  - Asserts that a reference is null.

- `assertNotNull(String message, Object object)`
  - Many 'not' asserts exists.
  - Most asserts have an optional message that can be printed.

Other Helpful JUnit Features

- @BeforeClass
  - Run once before all test methods in class.
- @AfterClass
  - Run once after all test methods in class.

- Together, these methods are used for setting up computationally expensive test elements.
  - E.g., database, file on disk, network…

- @Before
  - Run before each test method.
- @After
  - Run after each test method.

- Make tests independent by setting and resetting your testing environment.
  - E.g., creating a fresh object

- @Test(expected=ParseException.**class**)
  - When you expect an exception

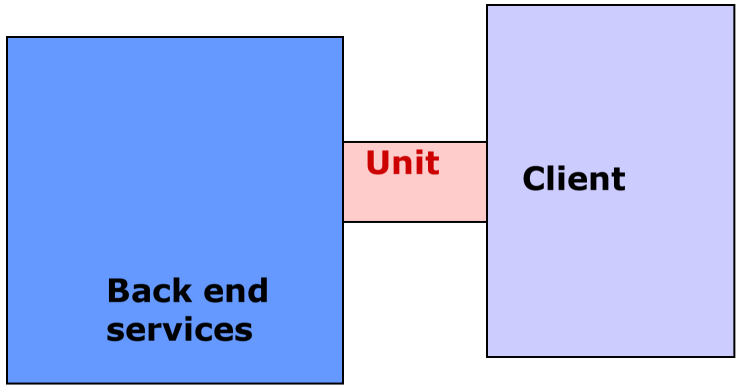
C Unit Testing Frameworks

- Check - http://check.sourceforge.net/
  - Full-featured unit-testing framework
  - Text output, cross-platform, isolates tests
- CUnit - http://cunit.sourceforge.net/
  - Full-featured unit-testing framework
  - Text output (graphics in Unix), cross-platform
- CuTest - http://cutest.sourceforge.net/
  - Basic unit-testing framework – super easy to get started
  - Text output, cross-platform
- cfix - http://cfix.sourceforge.net/
  - Full-featured unit-testing framework
  - Integrated with Microsoft tools

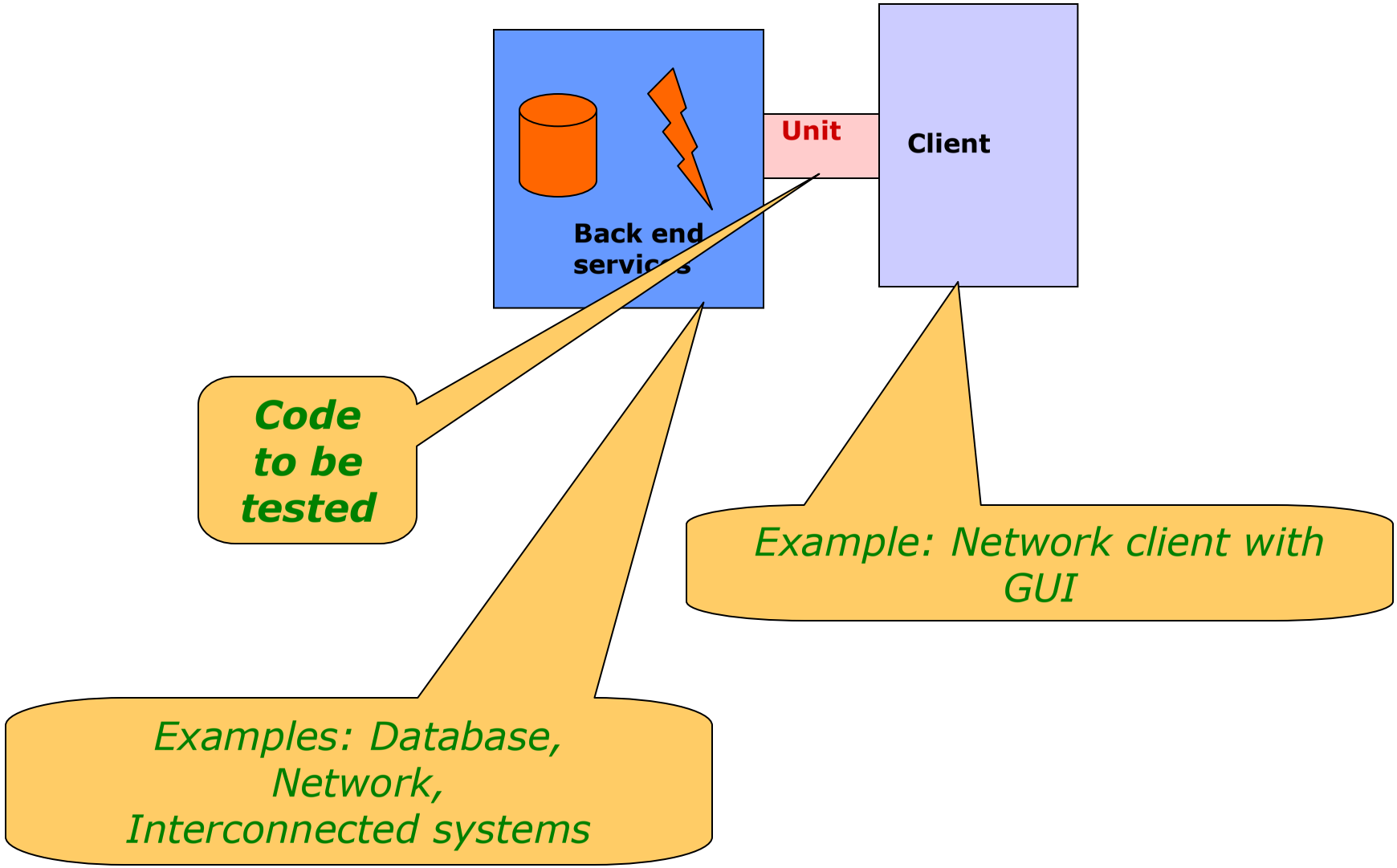
Testing – The Big Questions

1. **What is testing?**
   - And why do we test?
2. **To what standard do we test?**
   - Specification of behavior and quality attributes
3. **How do we select a set of good tests?**
   - Functional (black-box) testing
   - Structural (white-box) testing
4. **How do we assess our test suites?**
   - Coverage, Mutation, Capture/Recapture…
5. **What are effective testing practices?**
   - Levels of structure: unit, integration, system…
   - Design for testing: scaffolding
   - Effective testing 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*

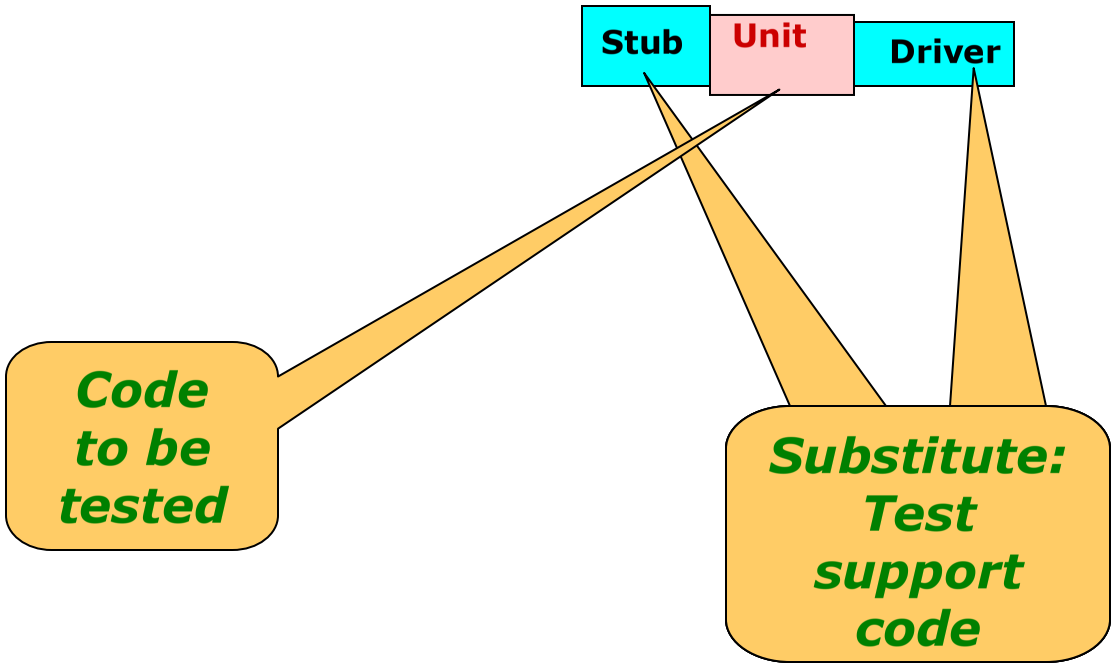
Unit Test and Scaffolding



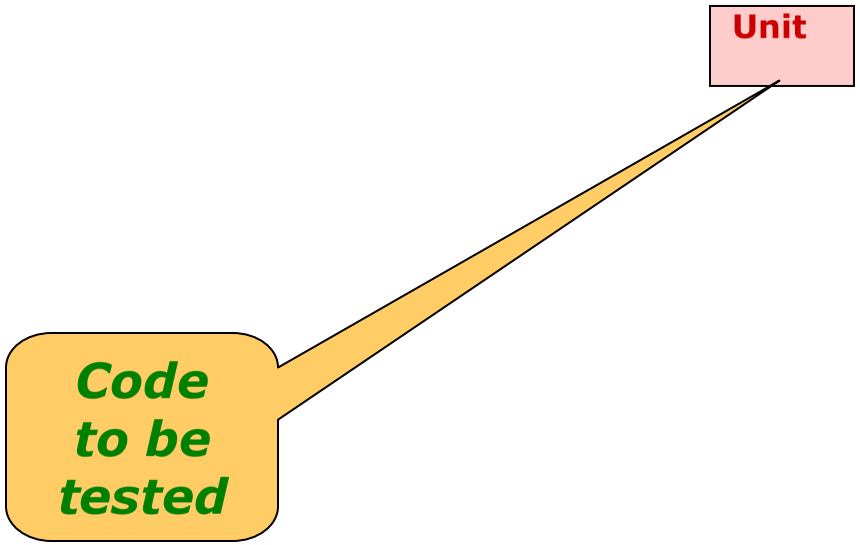
Unit Test and Scaffolding



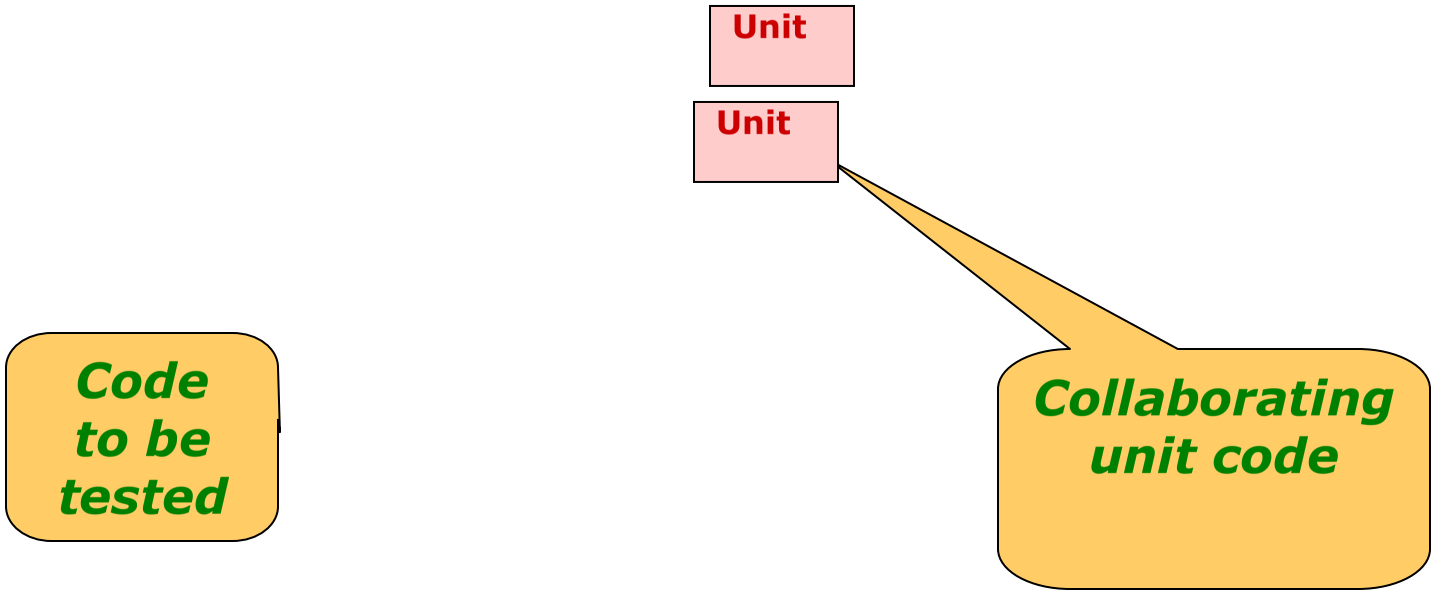
Unit Test and Scaffolding



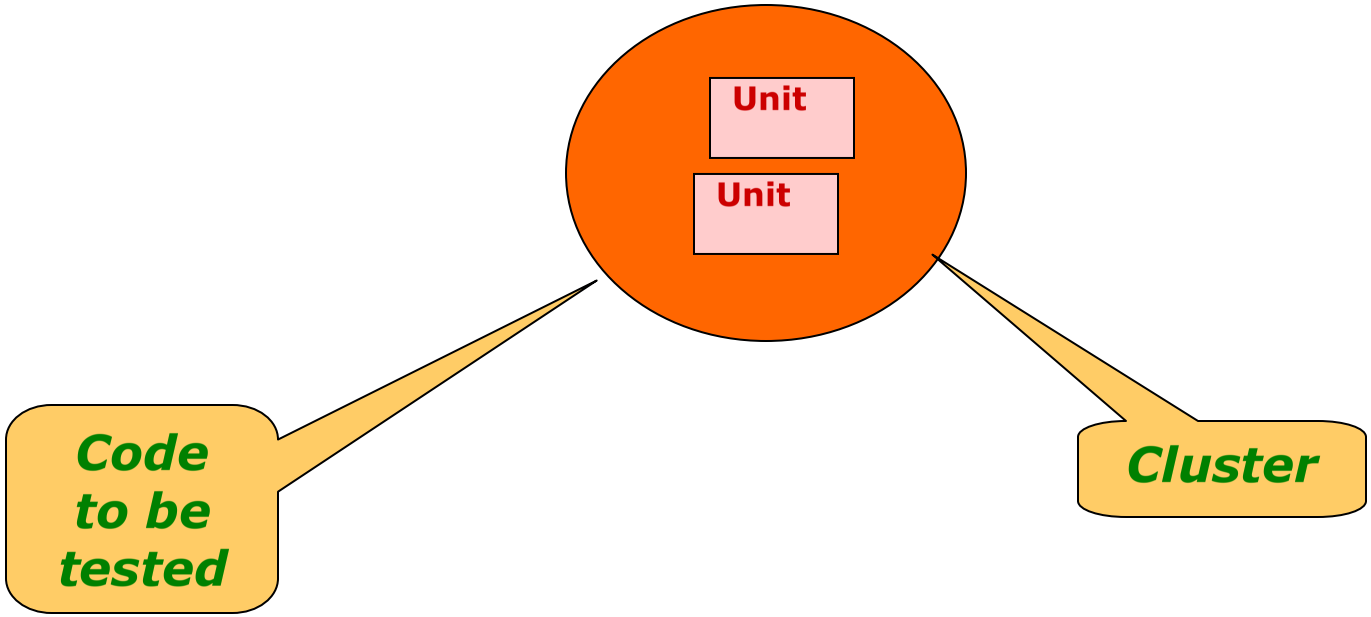
Unit Test and Scaffolding



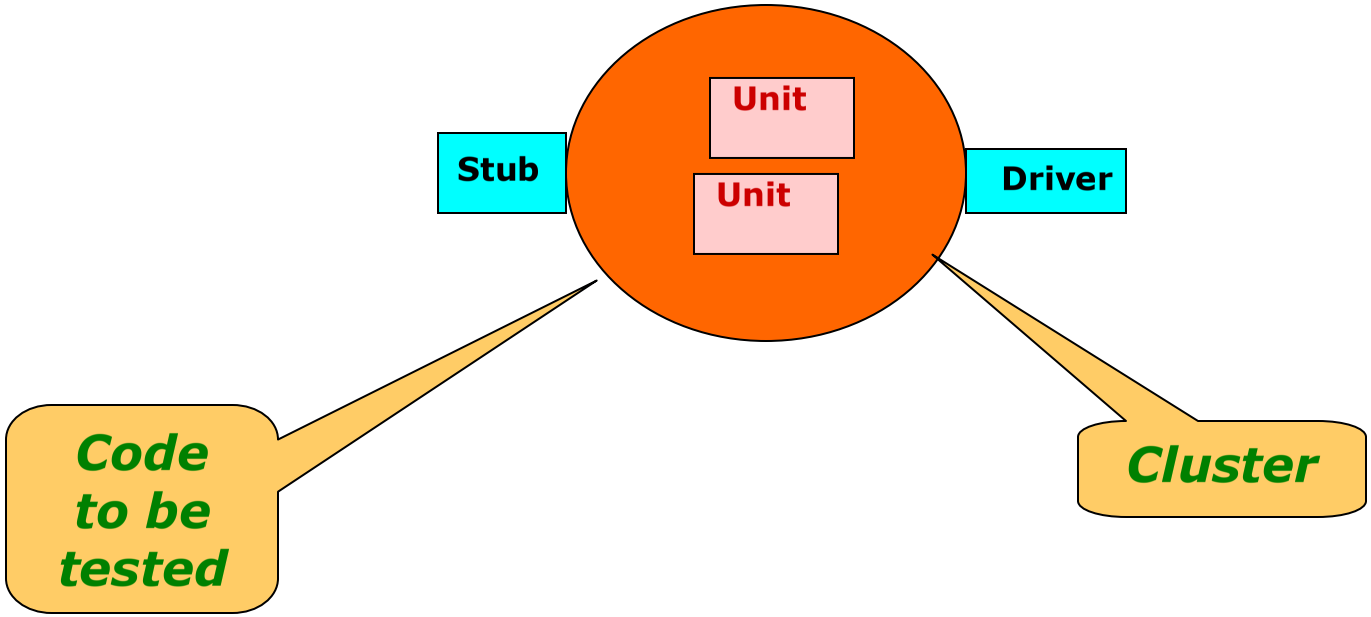
Unit Test and Scaffolding



Unit Test and Scaffolding



Unit Test and Scaffolding

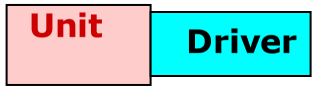


Techniques for Unit Testing 1: Scaffolding

- Use "scaffold" to simulate external code
- External code – scaffold points
  1. Client code
  2. Underlying service code

1. 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

Testers write driver code



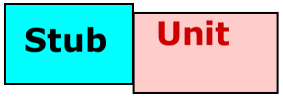
Techniques for Unit Testing 1: Scaffolding

- Use "scaffold" to simulate external code

- External code – scaffold points
  1. Client code
  2. Underlying service code

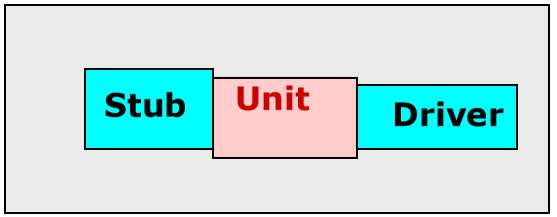
2. Service code
   - Underlying services
     - Communication services
       - Model behavior through a communications interface
       - Database queries and transactions
     - Network/web transactions
     - Device interfaces
       - Simulate device behavior and failure modes
     - File system
       - 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

Testers write stub code



Scaffolding

- 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"



Barriers to Scaffolding

- For some applications scaffolding is difficult
  - Wide interface between components
    - Must replicate entire interface
    - Automated tools can help
  - Complex behavior exercised in tests
    - Actual implementation may be simpler than scaffolding
    - Scaffolding may not be worthwhile here!
  - May be difficult to set up data structure for tests
    - Design principle - create special constructors for testing

Testing – The Big Questions

1. **What is testing?**
   - And why do we test?
2. **To what standard do we test?**
   - Specification of behavior and quality attributes
3. **How do we select a set of good tests?**
   - Functional (black-box) testing
   - Structural (white-box) testing
4. **How do we assess our test suites?**
   - Coverage, Mutation, Capture/Recapture…
5. **What are effective testing practices?**
   - Levels of structure: unit, integration, system…
   - Design for testing
   - Effective testing 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*

5c. Integration/System Testing

1. 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

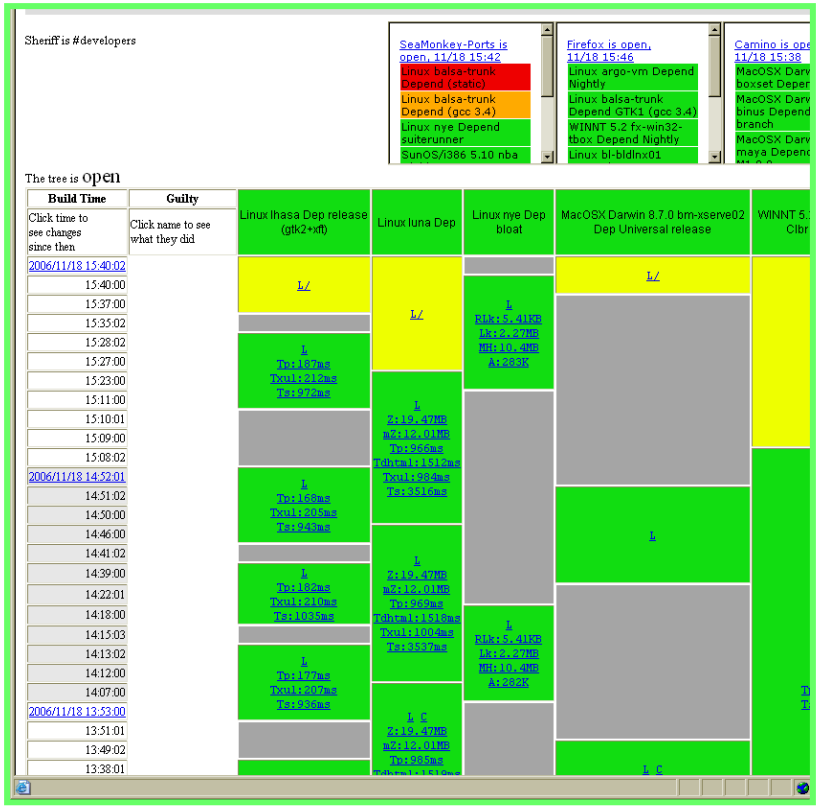
- Used focused tests
  - Set up subsystem for test
  - Test specific subsystem- or system-level features
    - no "random input" sequence
  - Verify expected output

5c. Frequent (Nightly) Builds

2. 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

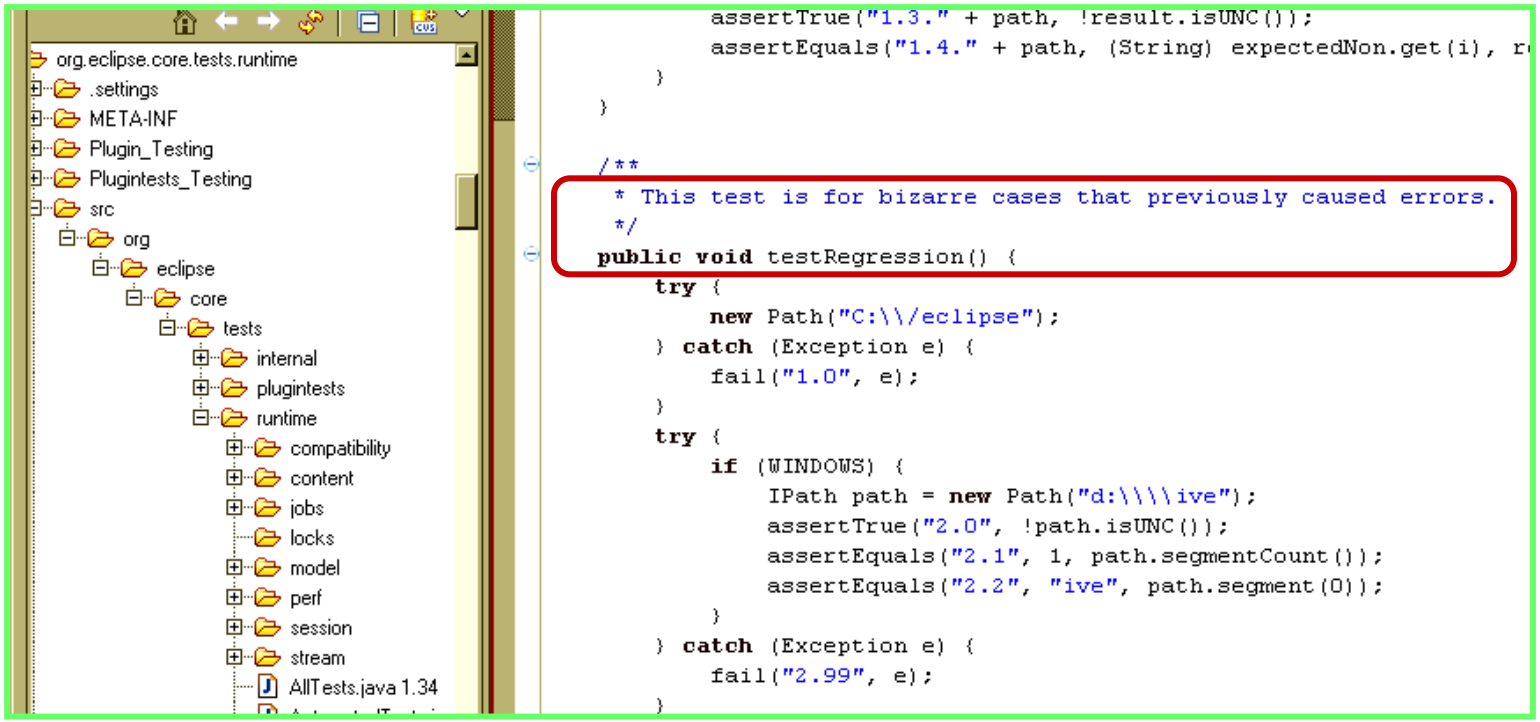


Practices –Regressions

3. 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

4. Acceptance tests (by customer)
   - Tests used by customer to evaluate quality of a system
   - Typically subject to up-front negotiation

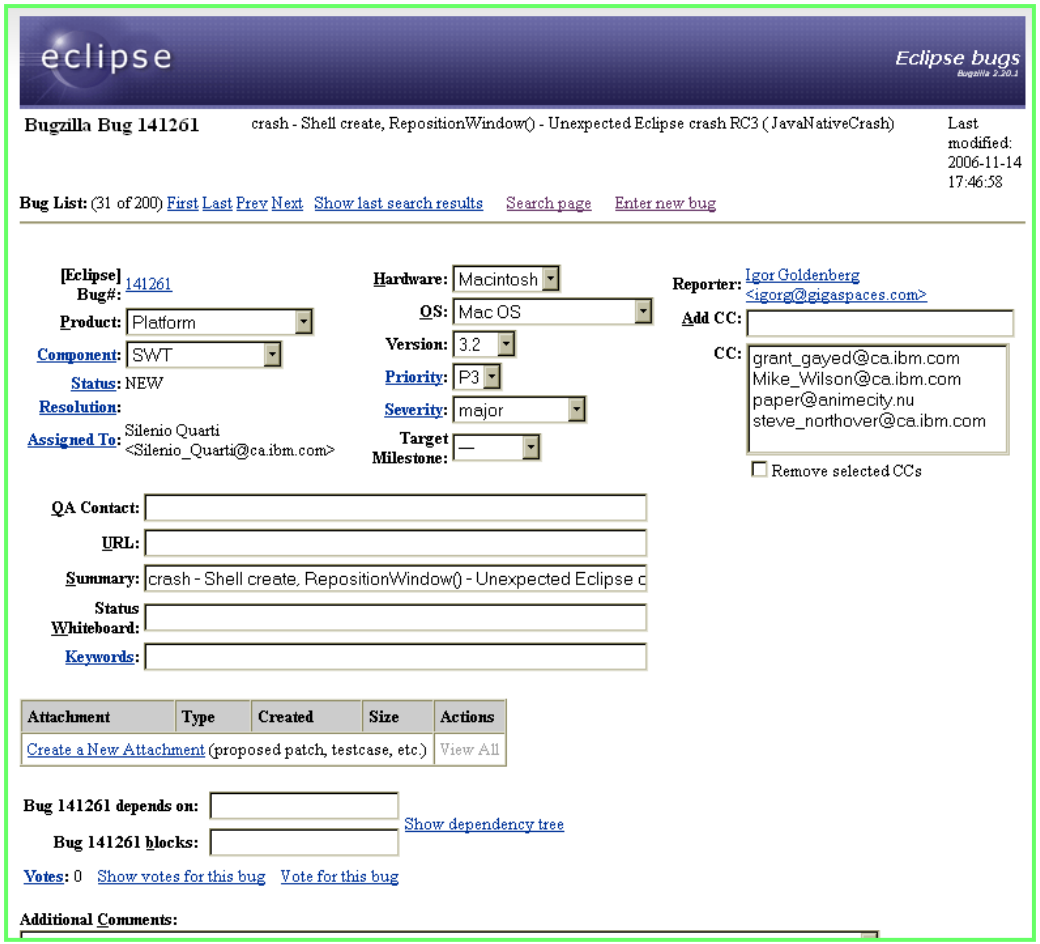
5. Release Test (by provider, vendor)
   - Test release CD
     - Before manufacturing!
   - Includes configuration tests, virus scan, etc
   - Carry out entire install-and-run use case

6. 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

7. Develop good defect reporting practices

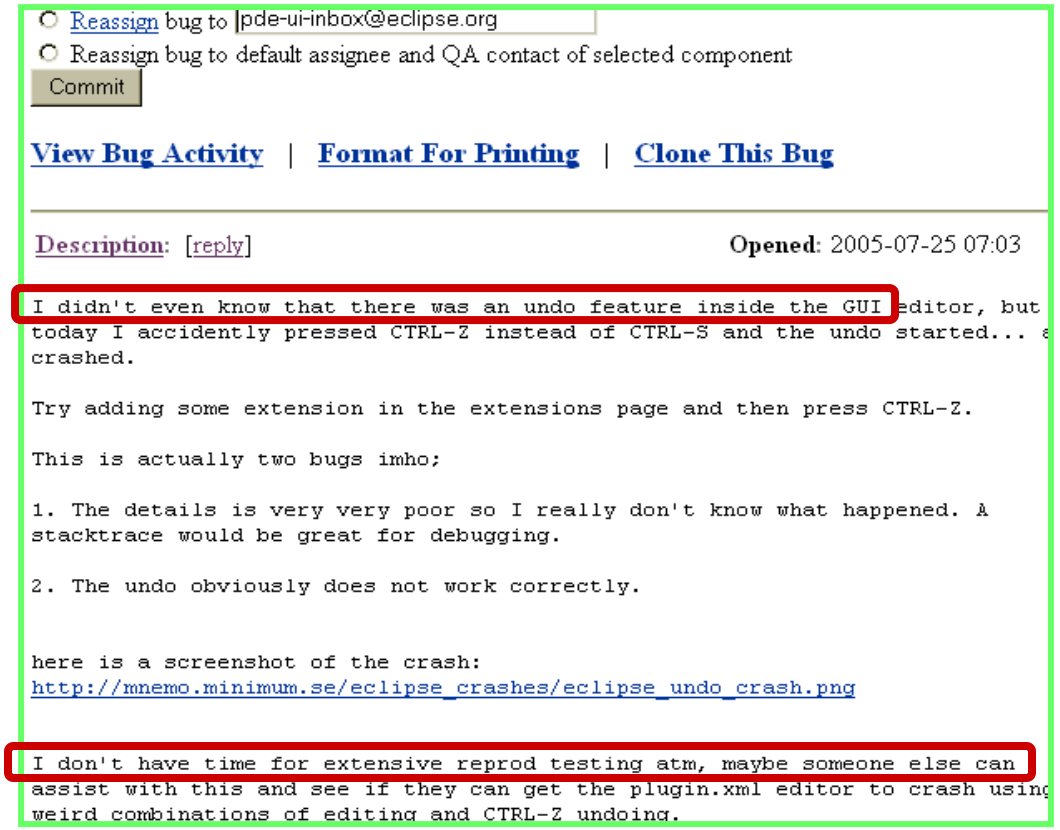
- Reproducible defects
  - Easier to find and fix
  - Easier to validate
    - Built-in regression test
  - Increased confidence
- Simple and general
  - More value doing the fix
  - Helps root-cause analysis
- Non-antagonistic
  - State the problem
  - Don't blame



Practices: Social Issues

8. 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



Practices: Root cause analysis

9. 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. **To what standard do we test?**
   - Specification of behavior and quality attributes
3. **How do we select a set of good tests?**
   - Functional (black-box) testing
   - Structural (white-box) testing
4. **How do we assess our test suites?**
   - Coverage, Mutation, Capture/Recapture…
5. **What are effective testing practices?**
   - Levels of structure: unit, integration, system…
   - Design for testing
   - Effective testing 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*

5d. 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

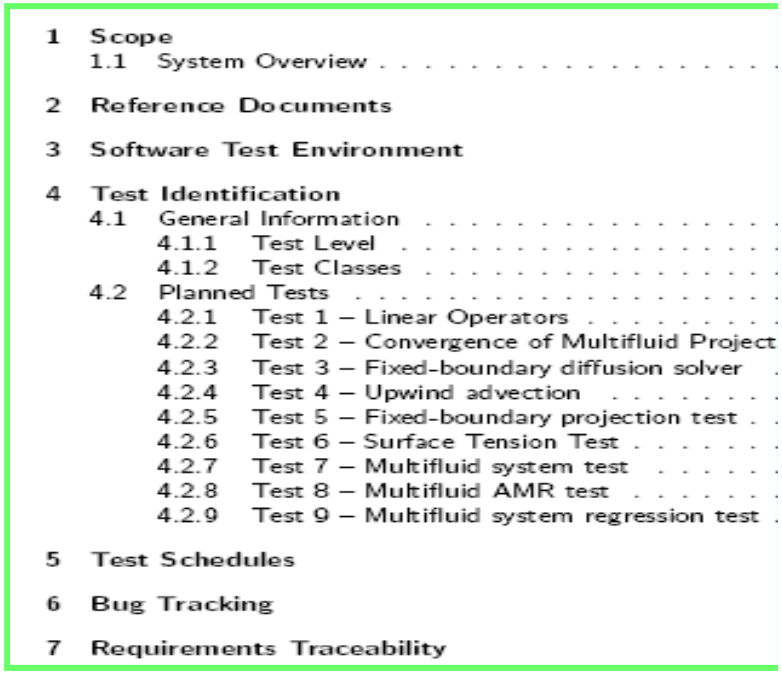
2. Favor unit testing over integration and system testing

- 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



- 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?

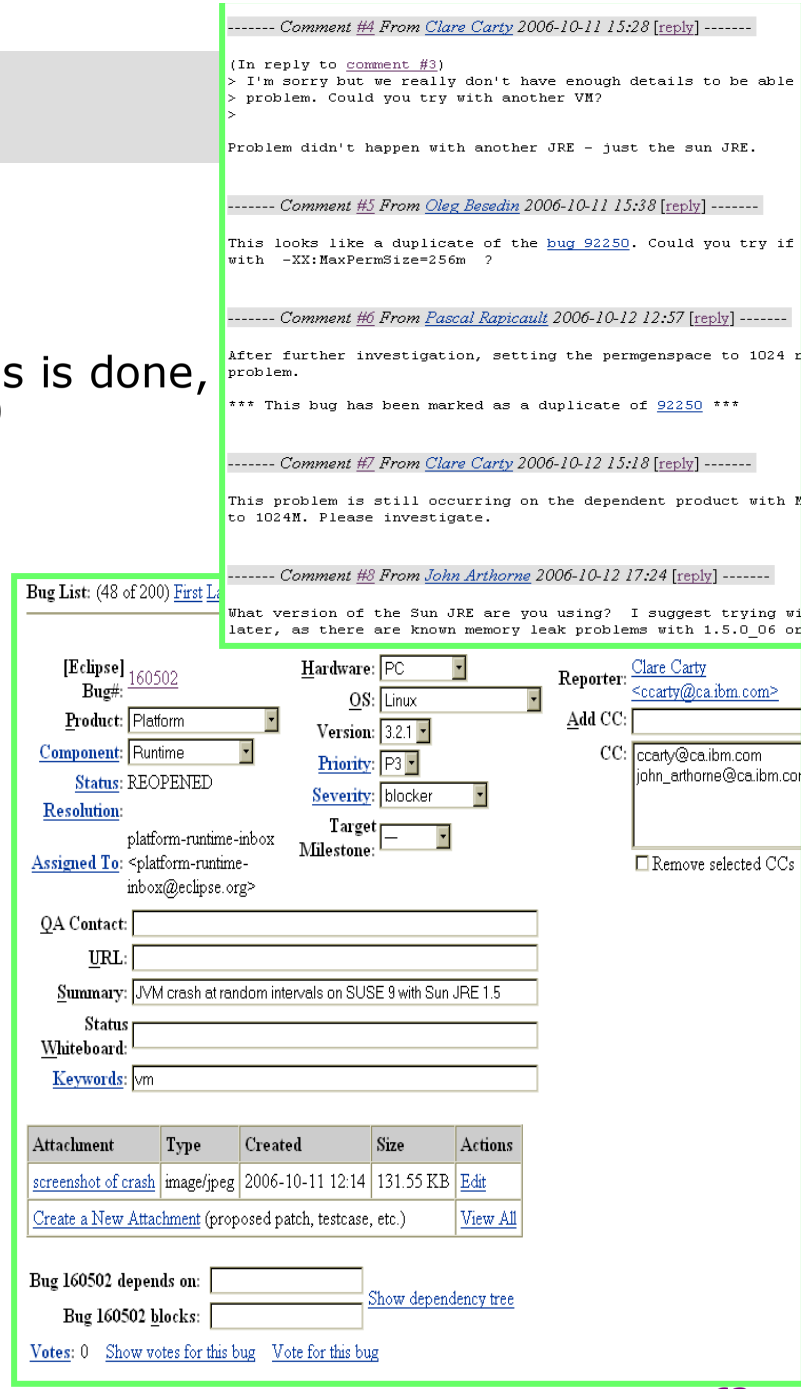
4. Ensure the test plan addresses the needs of stakeholders

- 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. **To what standard do we test?**
   - Specification of behavior and quality attributes
3. **How do we select a set of good tests?**
   - Functional (black-box) testing
   - Structural (white-box) testing
4. **How do we assess our test suites?**
   - Coverage, Mutation, Capture/Recapture…
5. **What are effective testing practices?**
   - Levels of structure: unit, integration, system…
   - Design for testing
   - 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

- Formal verification
  - Hoare Logic – verification of functional correctness
  - ESC/Java - automated verification
- Static analysis: provable correctness
  - Reflexion models, ArchJava – conformance to design
  - Fluid – concurrency analysis for race conditions
  - Plural – 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
- Analysis Across the Software Lifecycle
  - Security Development Lifecycle – architectural analysis for security
  - Design Structure Matrices – evolvability analysis
  - Alloy – systematically exploring a model of a design
  - Process analysis – defect prediction, schedule analysis, …

Questions?