Testing Retrospective

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

Jonathan Aldrich
Assistant Professor
Institute for Software Research

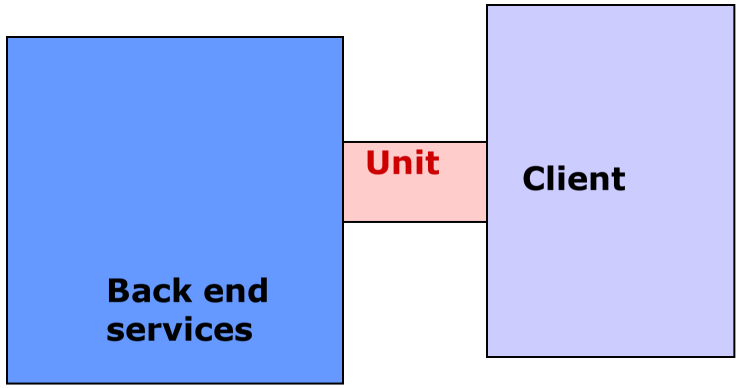
School of Computer Science
Carnegie Mellon University

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

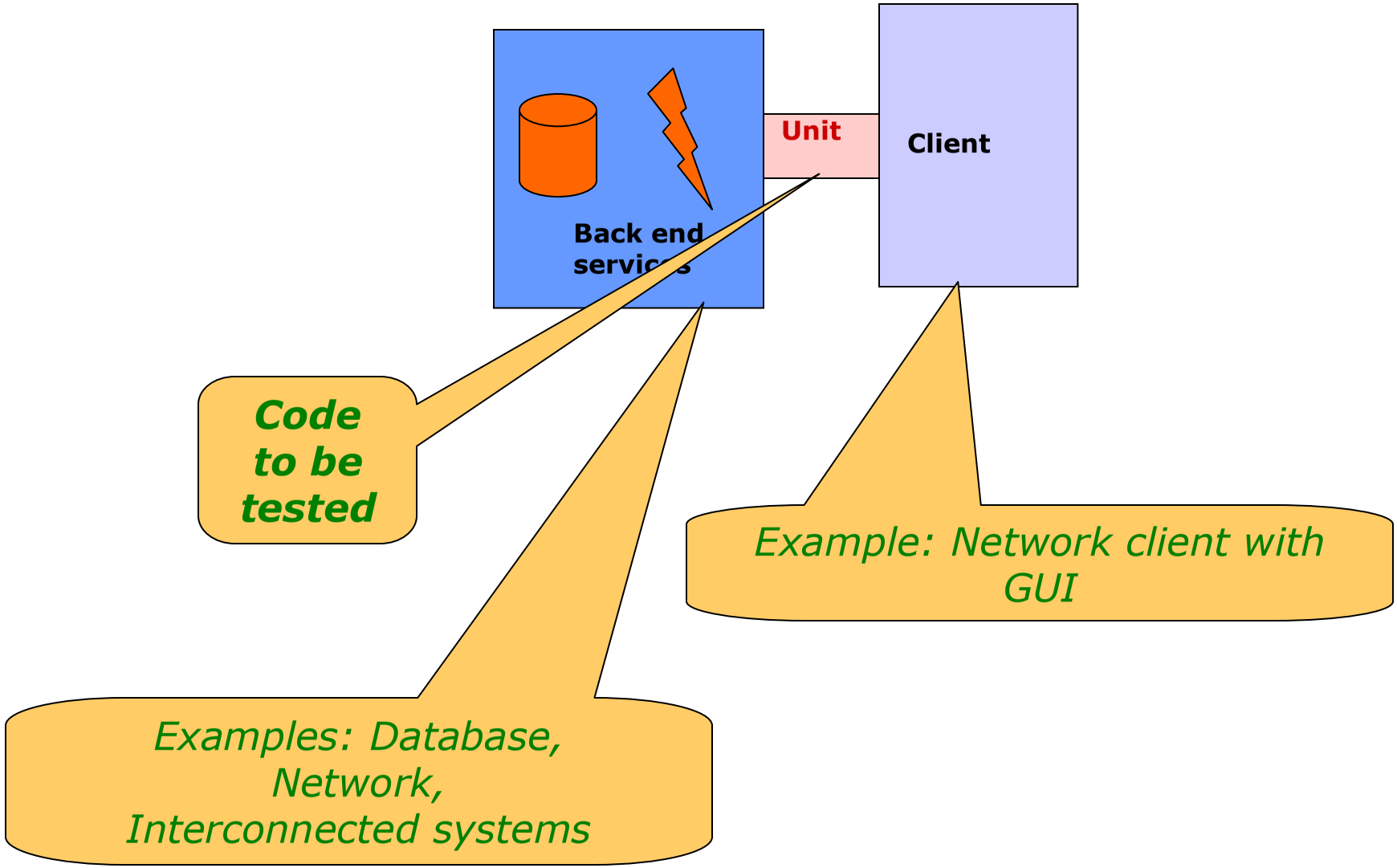
Testing – The Big Questions

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*

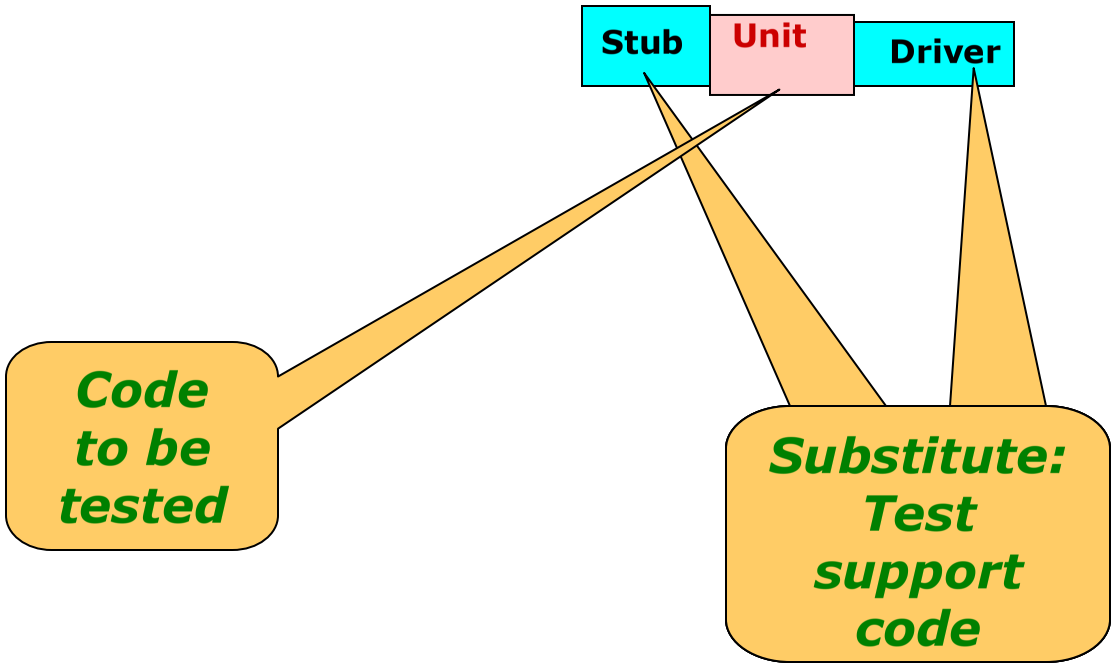
Unit Test and Scaffolding



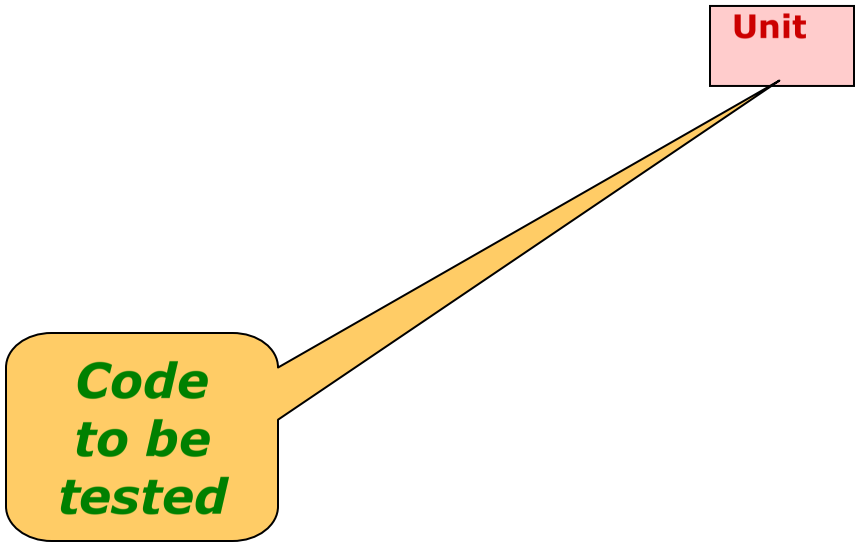
Unit Test and Scaffolding



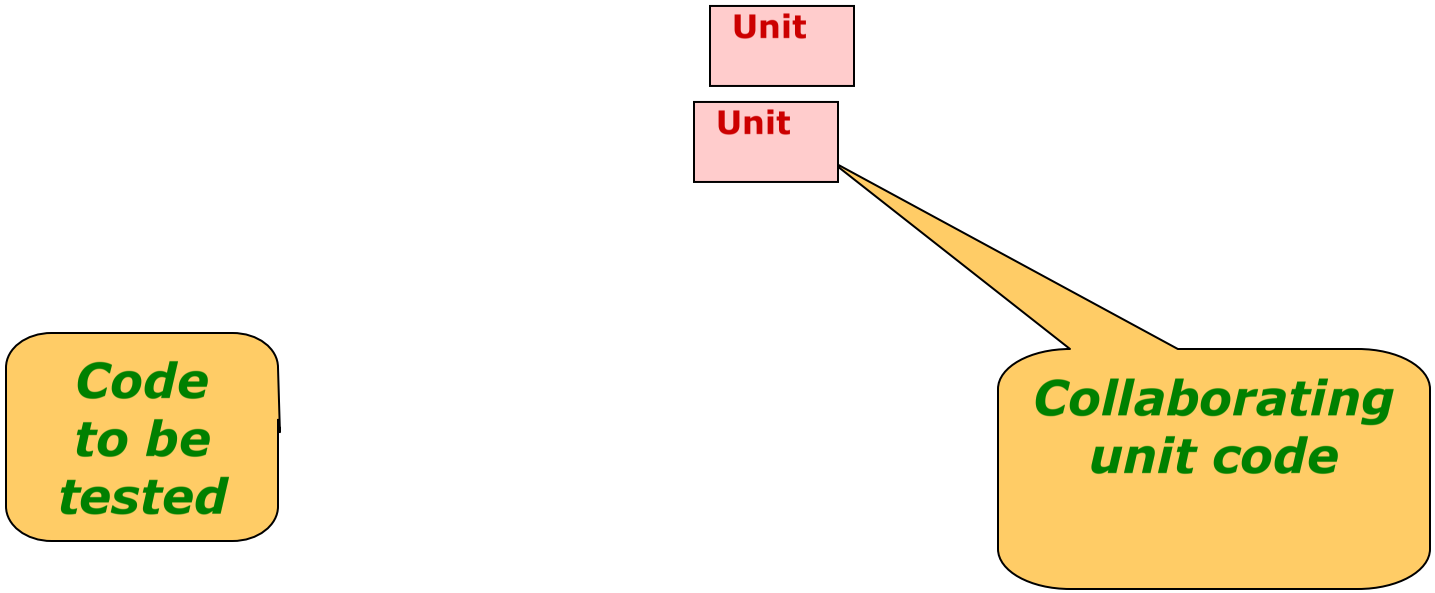
Unit Test and Scaffolding



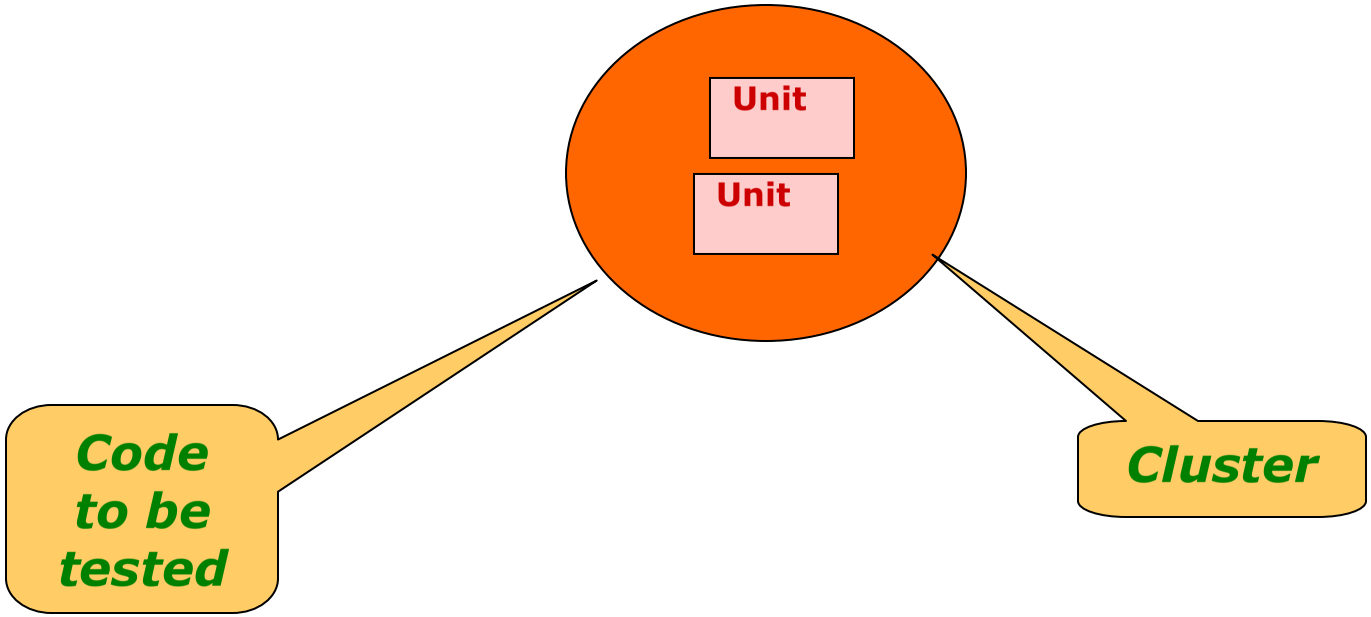
Unit Test and Scaffolding



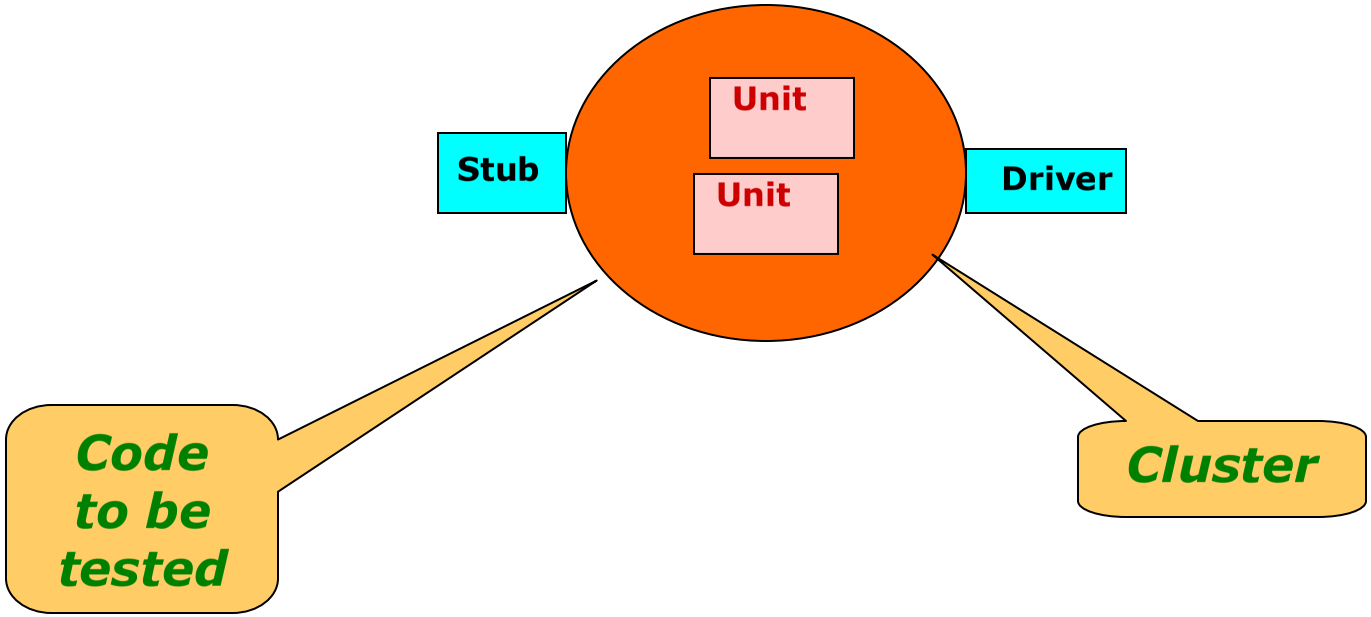
Unit Test and Scaffolding



Unit Test and Scaffolding



Unit Test and Scaffolding

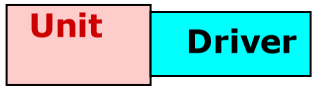


Techniques for Unit Testing 1: Scaffolding

- 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



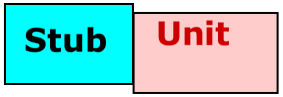
Techniques for Unit Testing 1: Scaffolding

- Use "scaffold" to simulate external code

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

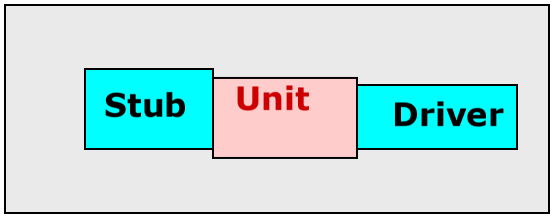
2. Service code
   - Underlying services
     - Communication services
       - Model behavior through a communications interface
       - Database queries and transactions
     - Network/web transactions
     - Device interfaces
       - Simulate device behavior and failure modes
     - File system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

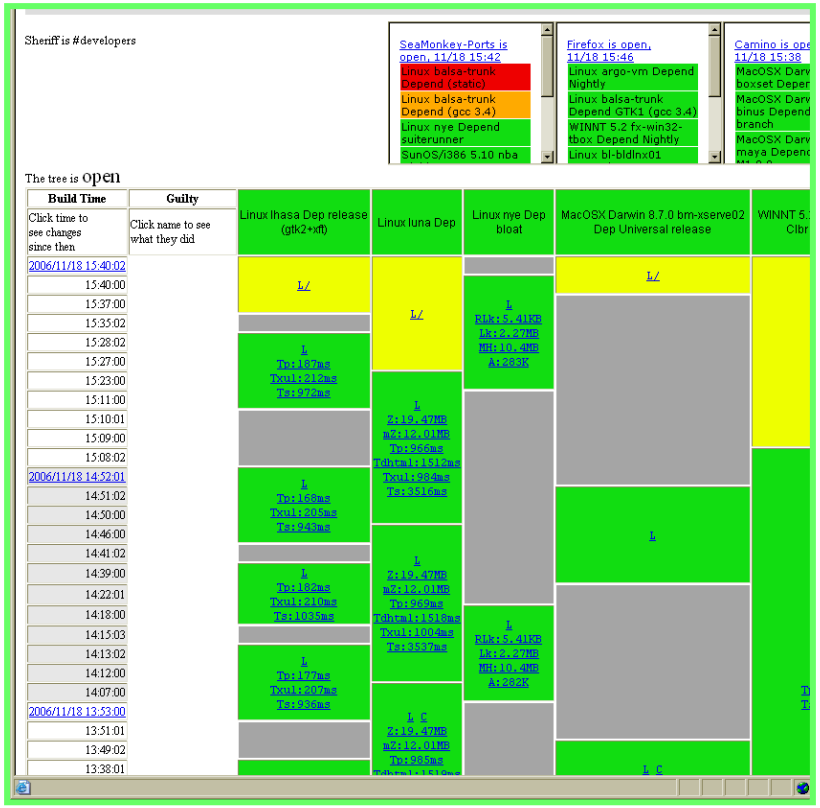
- Used focused tests
  - Set up subsystem for test
  - Test specific subsystem- or system-level features
    - no "random input" sequence
  - Verify expected output

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

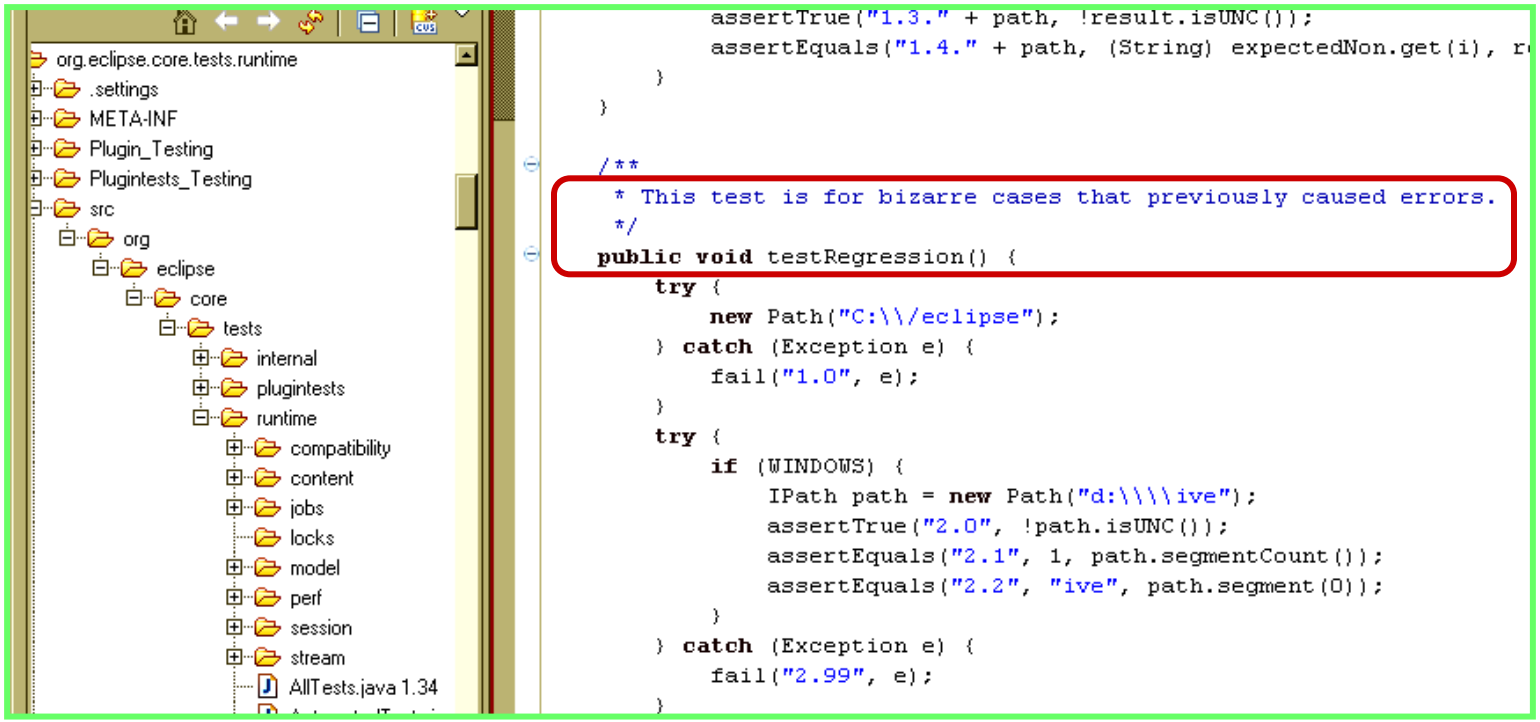


Practices –Regressions

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

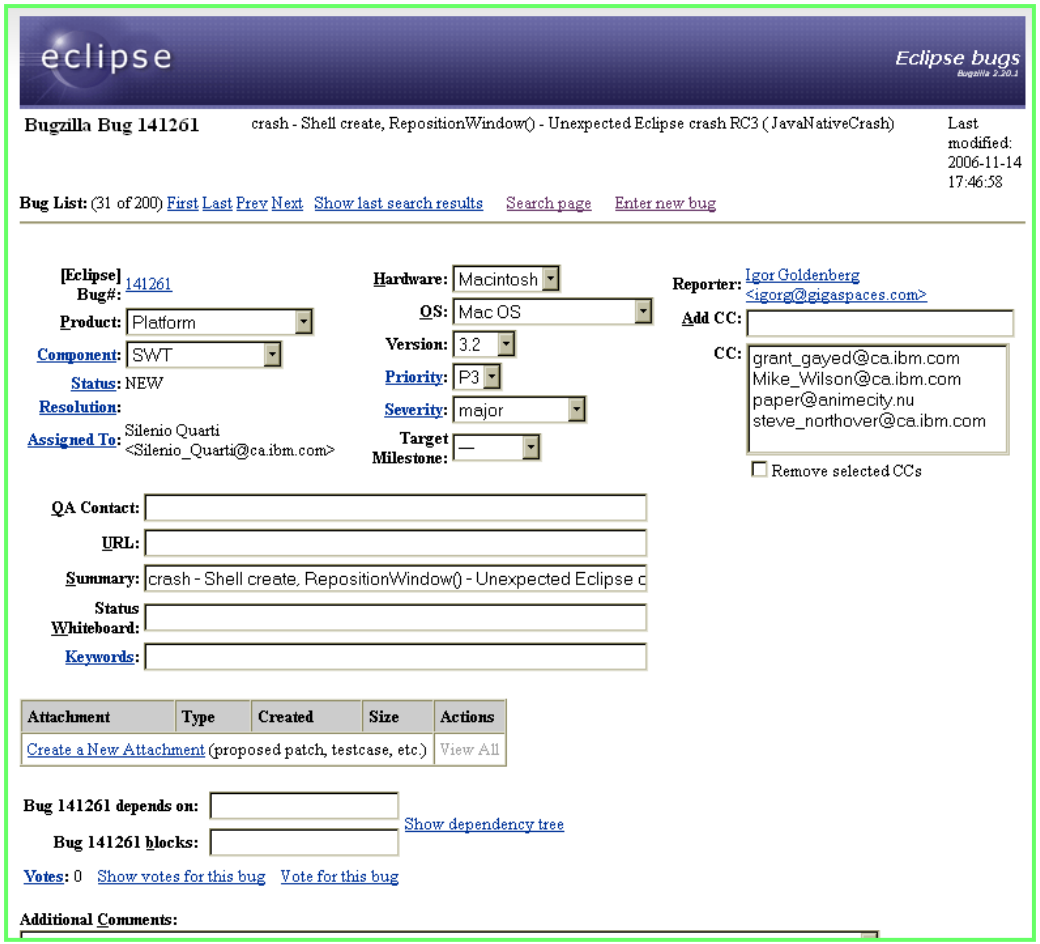
5. Release Test (by provider, vendor)
   - Test release CD
     - Before manufacturing!
   - Includes configuration tests, virus scan, etc
   - Carry out entire install-and-run use case

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

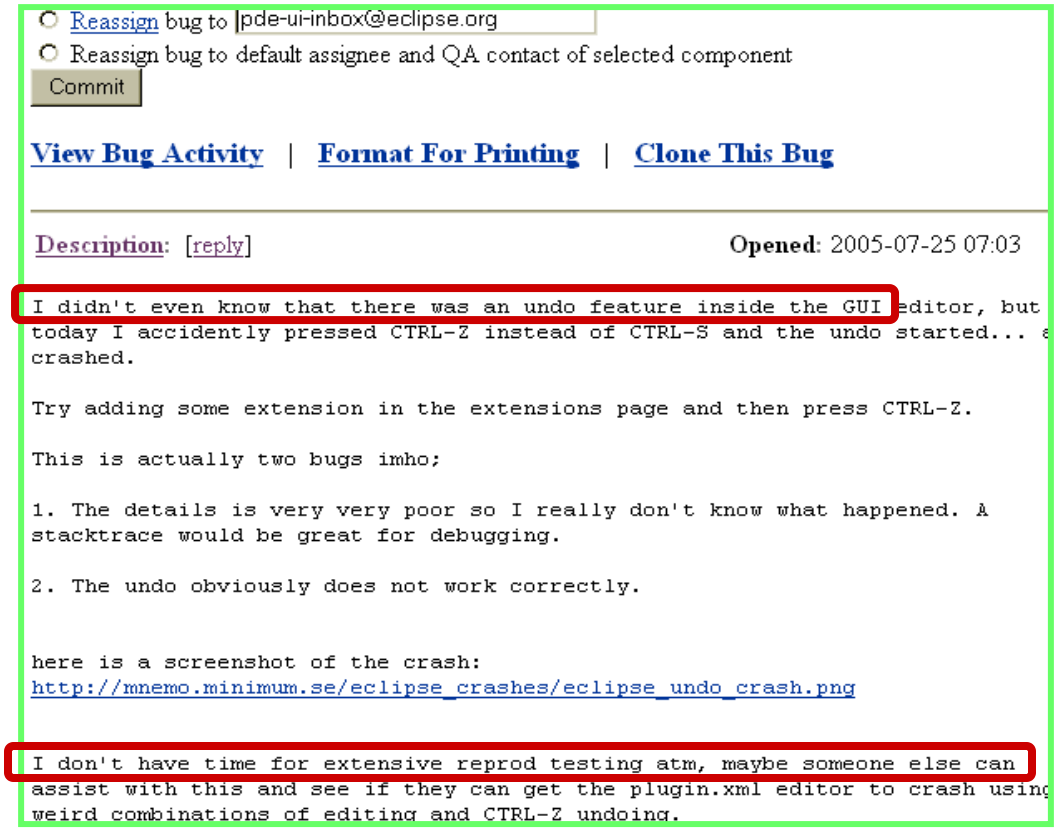
- Reproducible defects
  - Easier to find and fix
  - Easier to validate
    - Built-in regression test
  - Increased confidence
- Simple and general
  - More value doing the fix
  - Helps root-cause analysis
- Non-antagonistic
  - State the problem
  - Don't blame



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

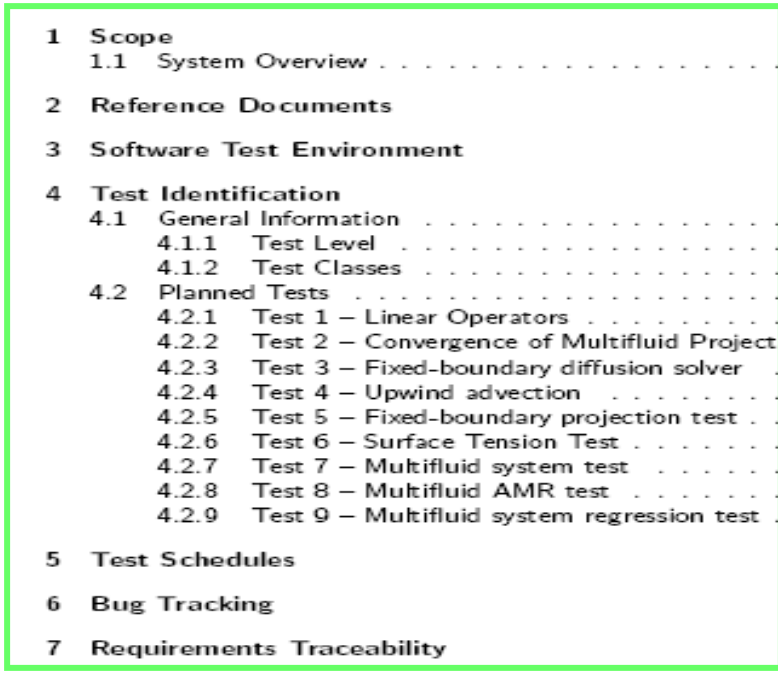
2. Favor unit testing over integration and system testing

- 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?

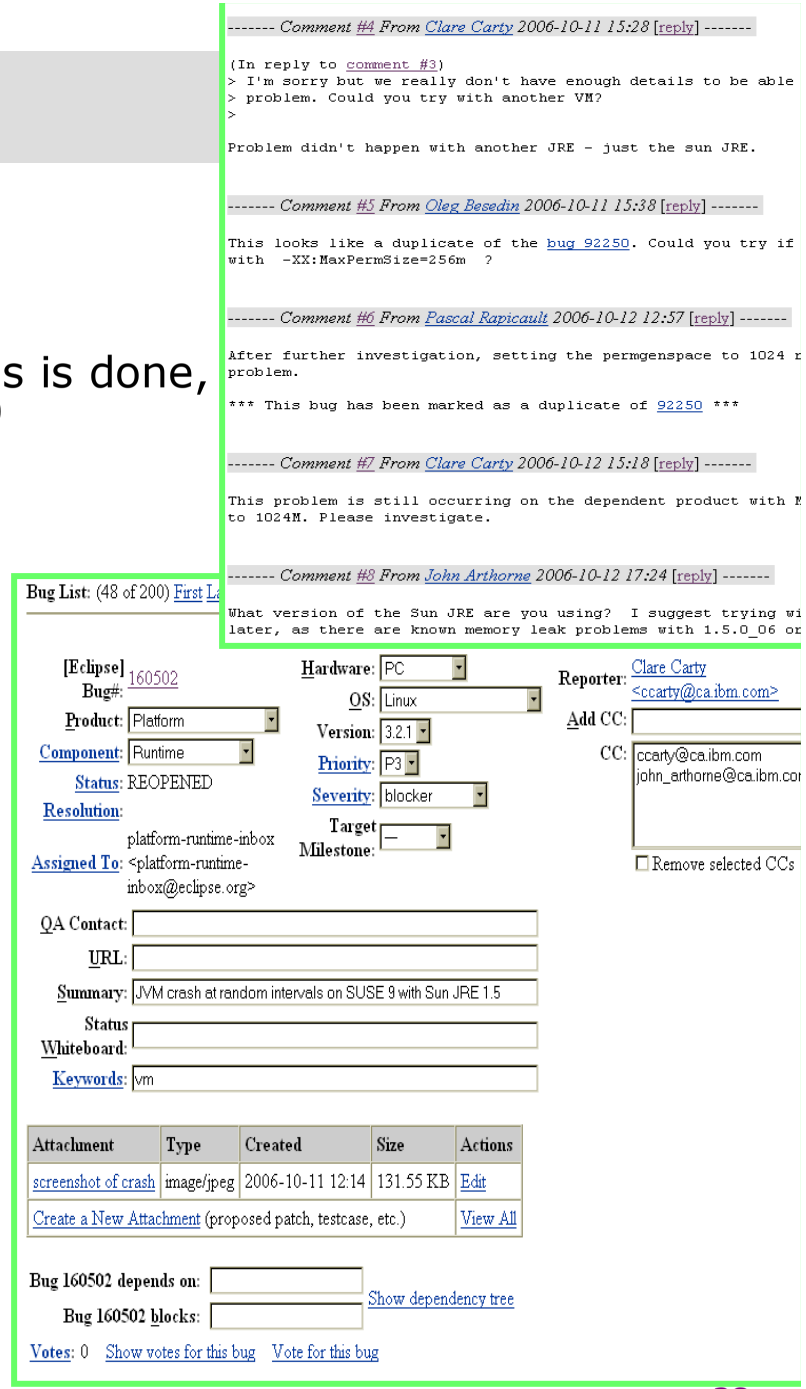
4. Ensure the test plan addresses the needs of stakeholders

- 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



Questions?