17-654/17-754
Analysis of Software Artifacts

Spring 2007

Testing

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. **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*

1. Testing: What and Why

- What is testing?
  - Direct execution of code on test data in a controlled environment

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

Testing is NOT to show correctness

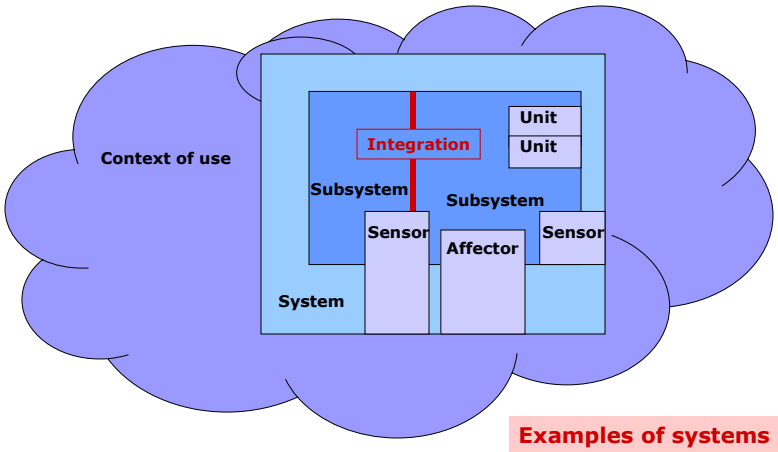
- Theory: "Complete testing" is impossible
  - For realistic programs there is always untested input
  - The program may fail on this input

- Psychology: Test to find bugs, not to show correctness
  - Showing correctness: you fail when program does
  - Psychology experiment
    - People look for blips on screen
    - They notice more if rewarded for finding blips than if penalized for giving false alarms
  - Testing for bugs is more successful than testing for correctness
    - [Teasley, Leventhal, Mynatt & Rohlman]

Testing – The Big Questions

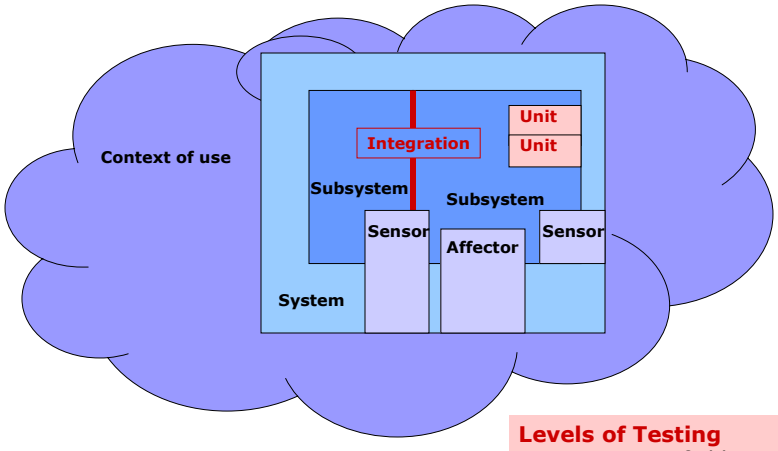
1. **What is testing?**
   - And why do we test?
2. **What do we test?**
   - Levels of structure: unit, integration, system…
3. **How do we select a set of good tests?**
   - Value-driven testing
   - Functional (black-box) testing
   - Structural (white-box) testing
4. **How do we assess our test suites?**
   - 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*

2. 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
- 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]

Contracts and Unit Tests

- A method's contract is a statement of the responsibilities of that method, and the responsibilities of the code that calls it.
  - Analogy: legal contracts
    - If you pay me exactly $30,000
    - I will build a new room on your house
  - Helps to pinpoint responsibility

- Examples:

```
/** Applies a move to a board. This assumes that the move is one that
   was returned by getAllMoves. Upon applying the move, it will also
   update the value of the board and switch the board's turn.  */
public void applyMove(Move mv) { … }


/*@ requires array != null
  @
  @ ensures \result == (\sum int j; 0<=j && j<array.length;  array[j])
  @*/
public float sum(int array[], int len) {… }
```

[source: Nels Beckman]

Objections to Unit Tests

- Objection: Writing unit tests takes too long
  - Must pay as you go, rather than pay at the end
- Answer: Unit tests raise productivity overall
  - Steady productivity throughout the development cycle
  - Without unit testing, productivity dives when testing starts
    - Must isolate bugs to their source
    - Must re-learn old code to debug it
    - Must often redesign code that was fundamentally broken

- Objection: We pay testers to write our tests
- Answer: Unit tests are for developers
  - Unit tests help you get your code working faster
    - There's egg on your face if you check in bad code – if you haven't tested it, how do you know?
    - Do you really want to be paid to spend hours with a debugger?
  - Testers still need to do functional, acceptance, user testing, etc.

- Unit tests help you even if you only do them yourself
  - But a culture of unit testing has additional benefits
    - regression tests, source control, integration testing

[source: Nels Beckman]

Unit Testing vs. printf, debuggers

- Can't you just use System.out.println?
  - Low-tech debug method
  - Output has to be scanned manually to see if it's correct
- Can't you just use a debugger?
  - A very manual process
  - Can't easily use it for regression

[source: Donna Malayeri]

Test-Driven Development

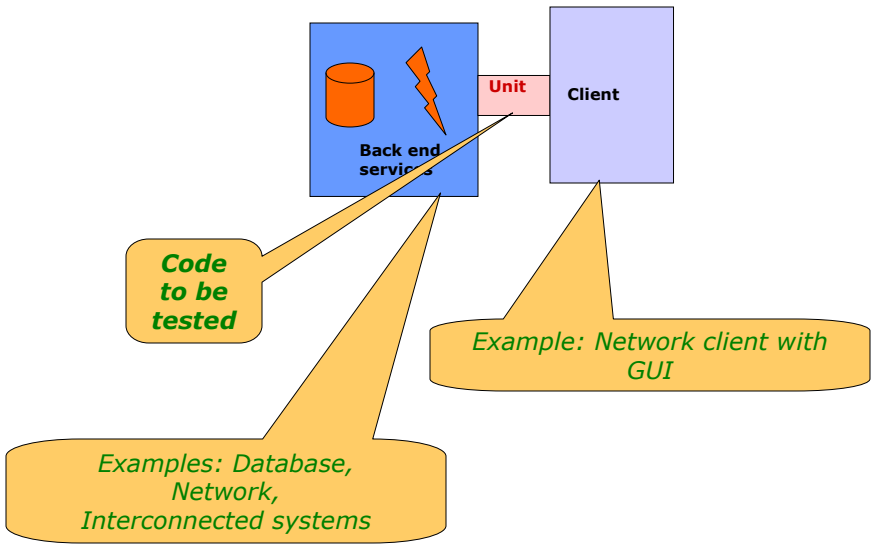
- Write the tests before the code
- Write code only when an automated test fails
- If you find a bug through other means, first write a test that fails, then fix the bug
  - Bug won't resurface later
- Run tests as often as possible, ideally every time the code is changed

[source: Donna Malayeri]

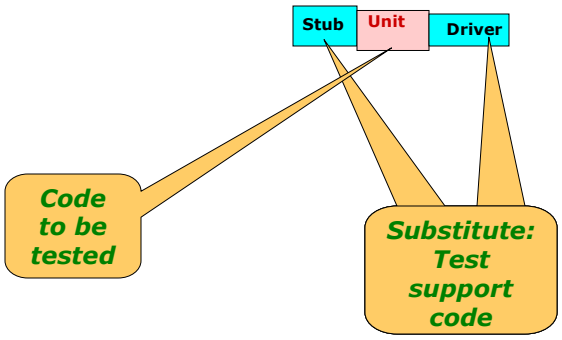
Unit Testing Pitfalls

- Random testing
  - Good for estimating quality when you know the input distribution
  - Bad for finding defects
- Writing tests without checking the output
  - Can find exceptions and crashes – but was the input valid?
  - Can check for output change – but was the original output right? was the change intended?
  - Moral: always check the input!
- Testing only valid inputs
  - Need to response to bad data (see especially security)
- Relying on code coverage for good testing
  - Bad coverage tells you your test suite is inadequate
  - But good coverage is not a guarantee of adequacy
    - Better coverage criteria: cover all important functional cases, borderline cases in the specification, and invalid inputs
      - More on these techniques later

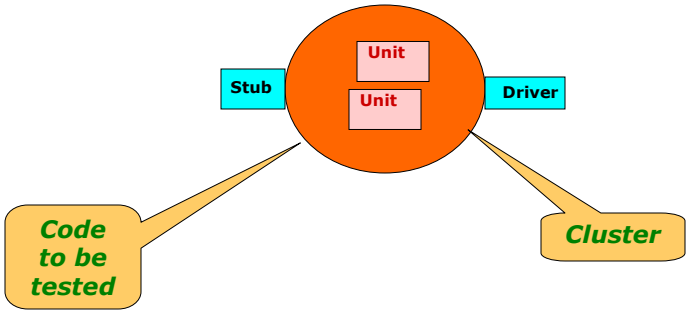
Unit Test and Scaffolding



Unit Test and Scaffolding



Unit Test and Scaffolding

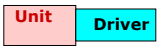


Techniques for Unit Testing 1: Scaffolding

- 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

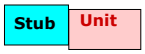


Techniques for Unit Testing 1: Scaffolding

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

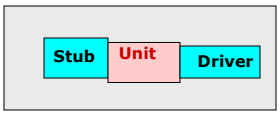
2. Service code
   - Underlying services
     - Communication services
       - Model behavior through a communications interface
       - Database queries and transactions
     - Network/web transactions
     - Device interfaces
       - Simulate device behavior and failure modes
     - File system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



Testing Harnesses

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

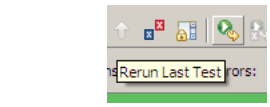
Help achieve three properties of good unit tests:

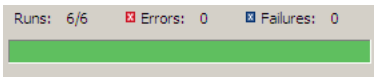
- Automatic
  - Tests should be easy to run and check for correct completion. This allows developers to quickly confirm their code is working after a change.
- Repeatable
  - Any developer can run the tests and they will work right away.
- Independent
  - Tests can be run in any order and they will still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

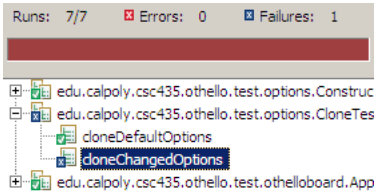
[source: Nels Beckman]

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

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*

Discussion: What makes a test case valuable?

- Value-driven testing
  - Focus on tests that have biggest benefit per unit cost
- Value is driven by quality improvement
  - Some value of information as well
- Value Factors
  - Does it find a bug?
  - How severe is the bug?
  - How common is the bug?
  - How easy is it to fix the bug?
  - Is it distinct from other tests?
    - Unique bug? Unique code? Unique domain coverage?
  - How general is it?
  - What did we learn about the program?
- Much of this is hard to predict in advance!

How do we select a set of good tests

- **Test coverage**
  - Why "coverage"?
    - All inputs cannot be tested.
  - Consider strategy for testing these systems:
    - Visual Studio, Eclipse, etc.
    - Automotive navigation/communication system – with many configurations
    - An operating system
    - An e-commerce container framework (J2EE, .net) and its components
  - Only very rarely can we test exhaustively.
    - Deterministic embedded controllers

Test coverage – Ideal and Real

- An **Ideal** Test Suite
  - Uncovers all errors in code
    - That are detectable through testing
  - Uncovers all errors in requirements capture
    - All scenarios covered
    - Non-functional attributes: performance, code safety, security, etc.
  - Minimum size and complexity
  - Uncovers errors early in the process
    - Ideally when code is being written ("test cases first")

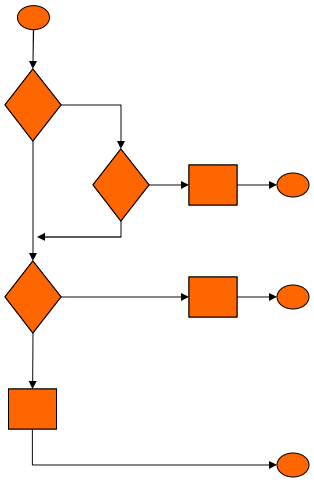
- A **Real** Test Suite
  - Uncovers some portion of errors in code
  - Has errors of its own
  - Assists in exploratory testing for validation
  - Does not help very much with respect to non-functional attributes
  - Includes many regression tests
    - Inserted after errors are repaired to ensure they won't reappear

Ways of analyzing coverage

- Code visibility –**white box** *or* **glass box**
  - Visibility to internal code elements – better for non-functional attributes
  - *Can use design information to guide creation and analysis of test suites*
  - Can test internal elements directly
  - **Code coverage analysis**
- Code visibility – **black box**
  - Cannot see internal code elements of the service being tested
  - Test through the public API – better for functional attributes
  - **Domain coverage analysis**

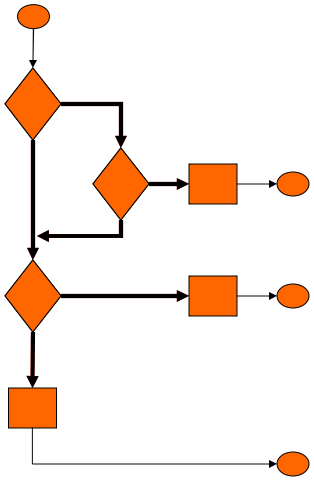
White Box: Statement Coverage

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



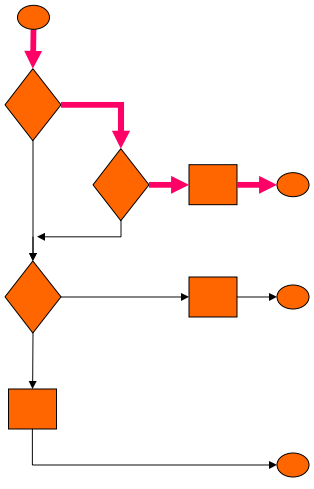
White Box: Branch Coverage

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



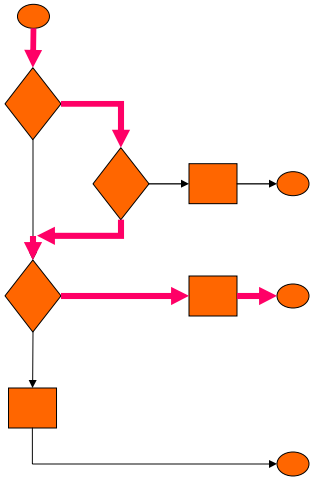
White Box: Path Coverage

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



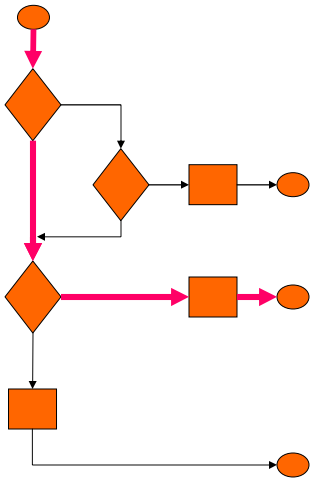
White Box: Path Coverage

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



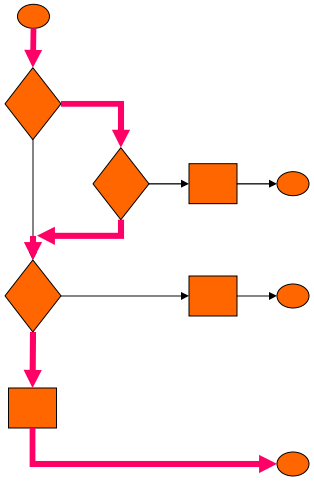
White Box: Path Coverage

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



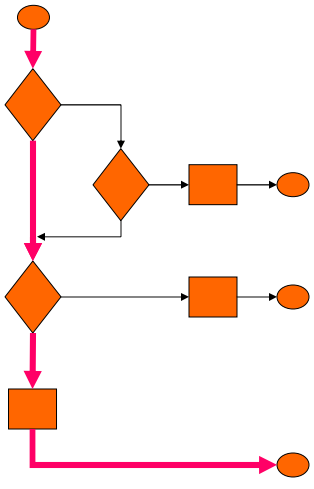
White Box: Path Coverage

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



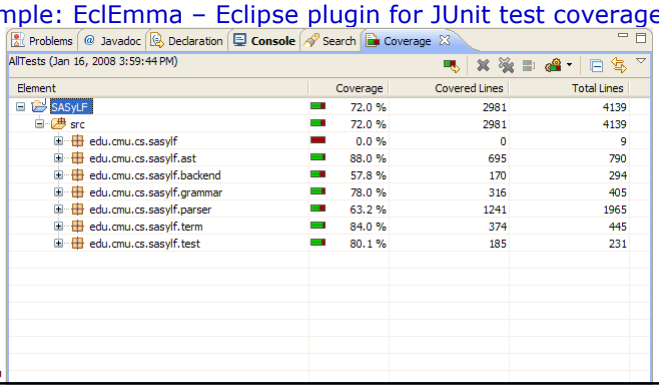
White Box: Path Coverage

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



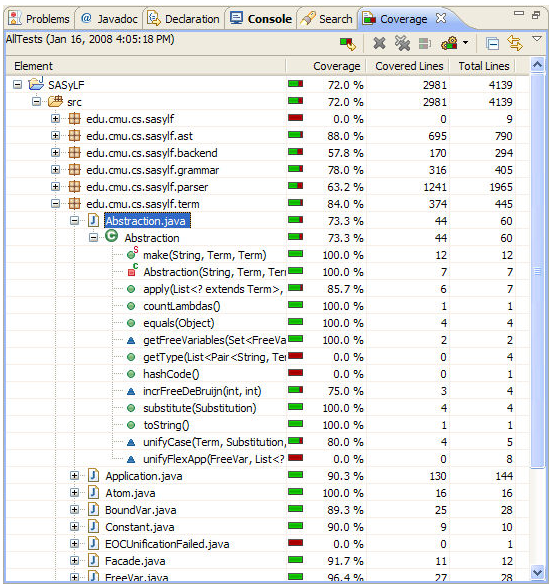
White Box: Assessing structural coverage

- Coverage assessment tools
  - Track execution of code by test cases
  - Techniques
    - Modified runtime environment (e.g., special JVM)
    - Source code transformation
- Count visits to statements
  - Develop reports with respect to specific coverage criteria
- Example: EclEmma – Eclipse plugin for JUnit test coverage



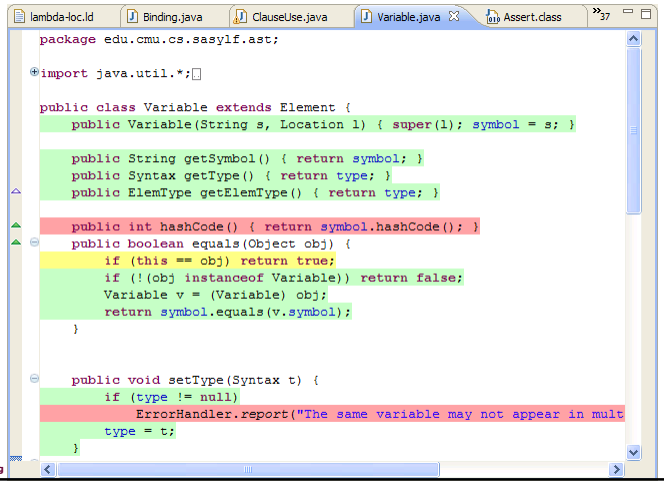
EclEmma in Eclipse

- Breakdown by package, class, and method
- Coverage
  - Classes
  - Methods
  - Statements
  - Instructions
- Graphical and numerical presentation



Clover in Eclipse

- Coverage report in editor window
  - red: not covered
  - yellow: covered once
  - green: covered multiple times



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
- Multiple white box coverage critieria
  - Useful to tell you where you are missing tests
  - Not sufficient to guarantee adequacy