Announcements

- Second assignment came out Monday evening
  - Topic: Code Inspection and Testing
    - Find defects in Hnefetafl rules written by your classmates
    - Compare inspection, coverage testing, random testing, and black-box testing
  - Group assignment
  - Due Wednesday, January 28, at 10:30am

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. **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*

Equivalence, Boundary and Robustness Example

- Example: Tic-Tac-Toe game
  - Two players, **X** and **O**, take turns marking the spaces in a 3×3 grid, with **X** going first. The player who succeeds in placing three respective marks in a horizontal, vertical or diagonal row wins the game.
- Equivalence classes?
- Boundary values?
- Robustness test cases?

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

Random Testing

- Randomly generate program input and execute test case
- Challenges
  - How to generate the input?
    - To assess quality, want representative input
    - To find defects, want good input coverage
  - How to check if the input is valid?
    - Check the precondition
    - If precondition false, throw out test or use as robustness test
  - How to check the output?
    - Need an *oracle* that can tell if the answer is right
      - Essentially the postcondition – but may not always be formally specified
    - May be easier to check an answer than to compute it
    - May have a reference implementation
    - May log information and check manually
    - May check only certain properties of the output
      - (partial) post-condition
      - lack of exceptions thrown

Checkpoints: Logging, Assertions, and Breakpoints

- 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

Assertions and Data Integrity Example

- What are some data integrity conditions on a Tic-Tac-Toe game position?

Test Coverage

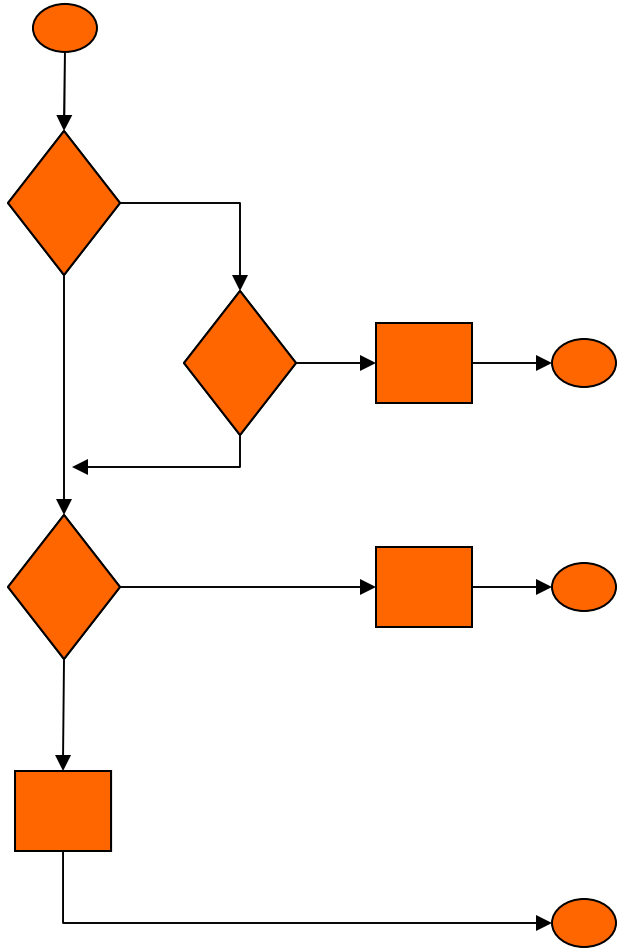
- Analysis: *the **systematic** examination of a software artifact to determine its properties*
  - We cannot test all inputs, so how can we be systematic?

- **Black box** testing
  - Systematically test different parts of the input domain
    - "domain coverage"
  - Test through the public API
    - focuses attention on client-visible behavior
  - No visibility into the code internals – a "black" box

- **White box** testing
  - Systematically test different elements in the code
    - "code coverage"
  - Can test internal elements directly – a "white" box
    - Good for quality attributes like robustness
  - Takes advantage of design information and code structure – a "white" box
    - "glass box" may be better terminology

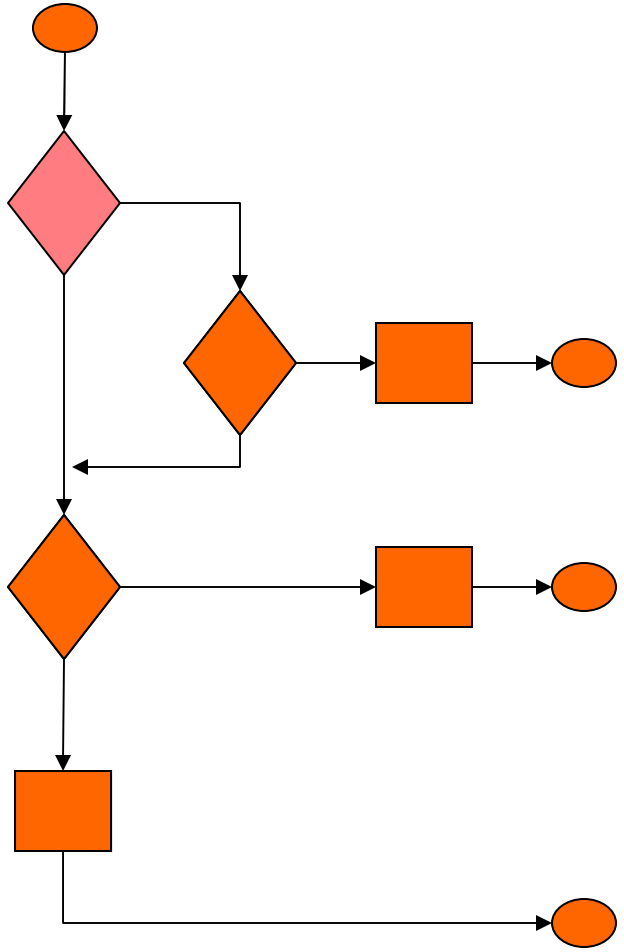
White Box: Statement Coverage

- **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!



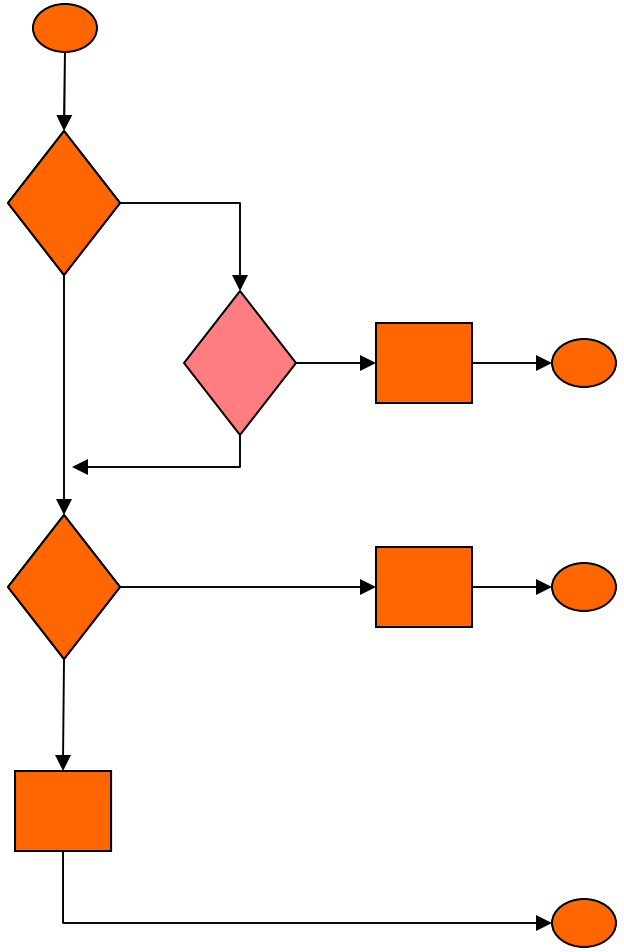
White Box: Statement Coverage

- **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!



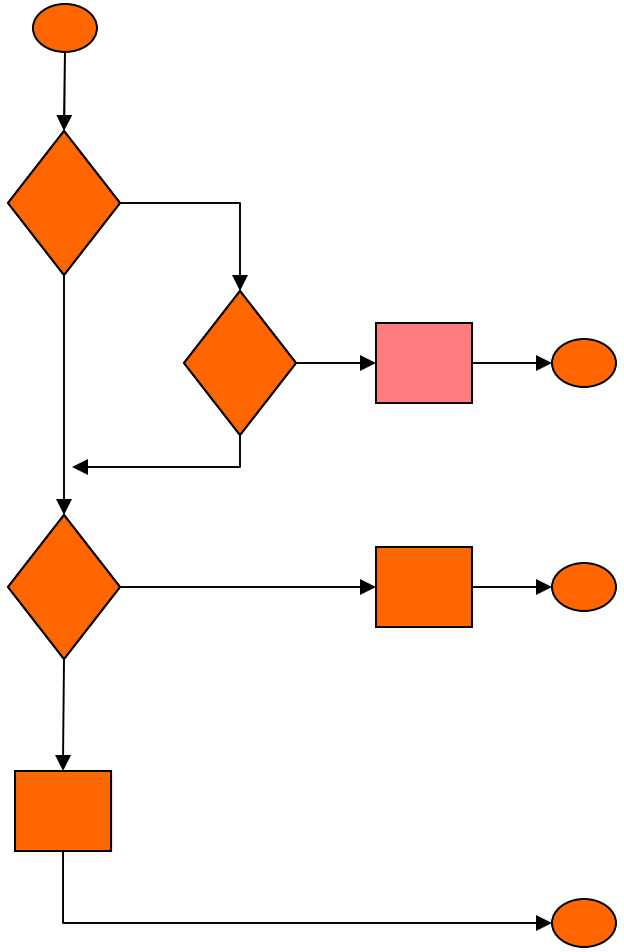
White Box: Statement Coverage

- **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!



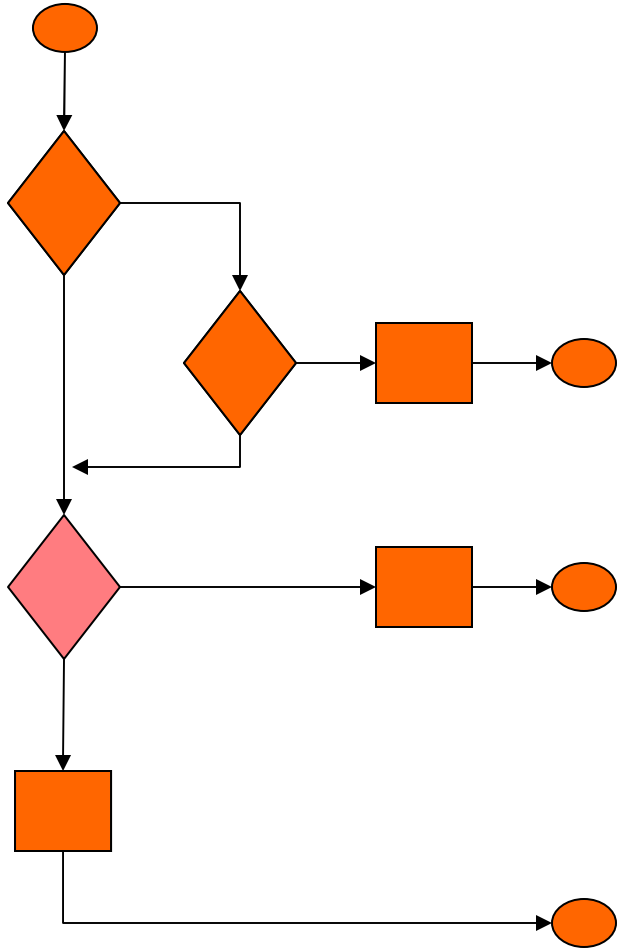
White Box: Statement Coverage

- **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!



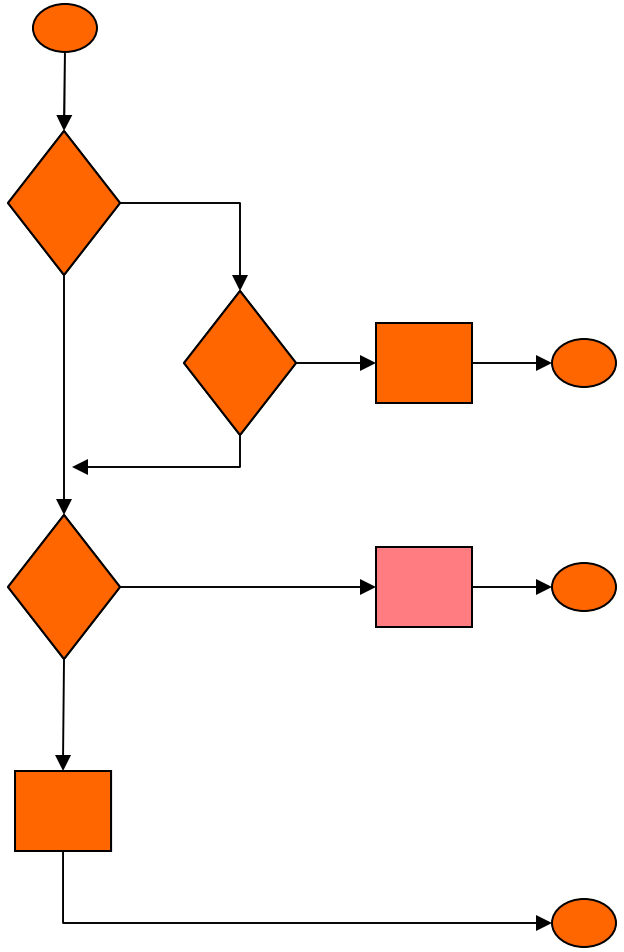
White Box: Statement Coverage

- **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!



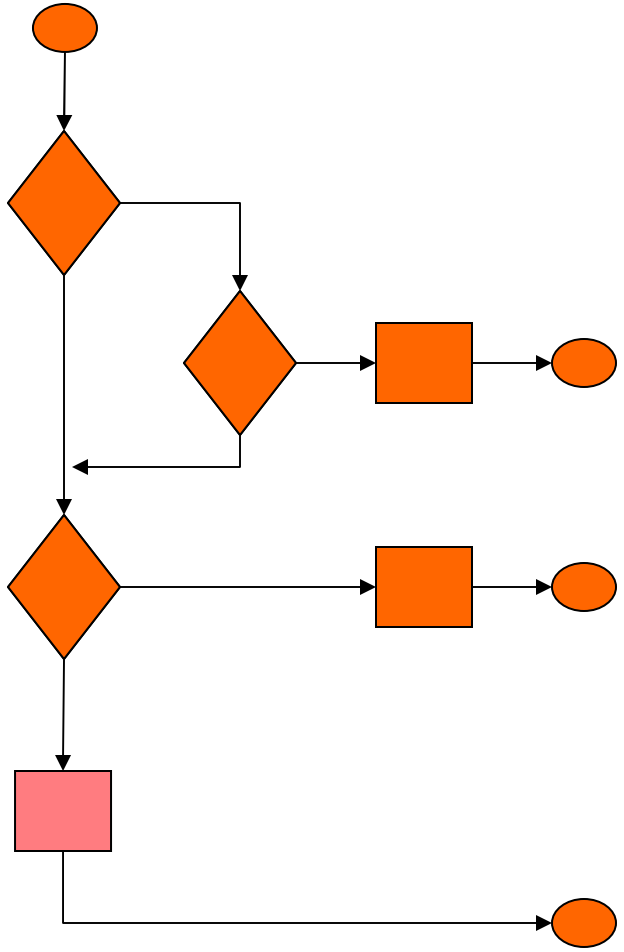
White Box: Statement Coverage

- **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!



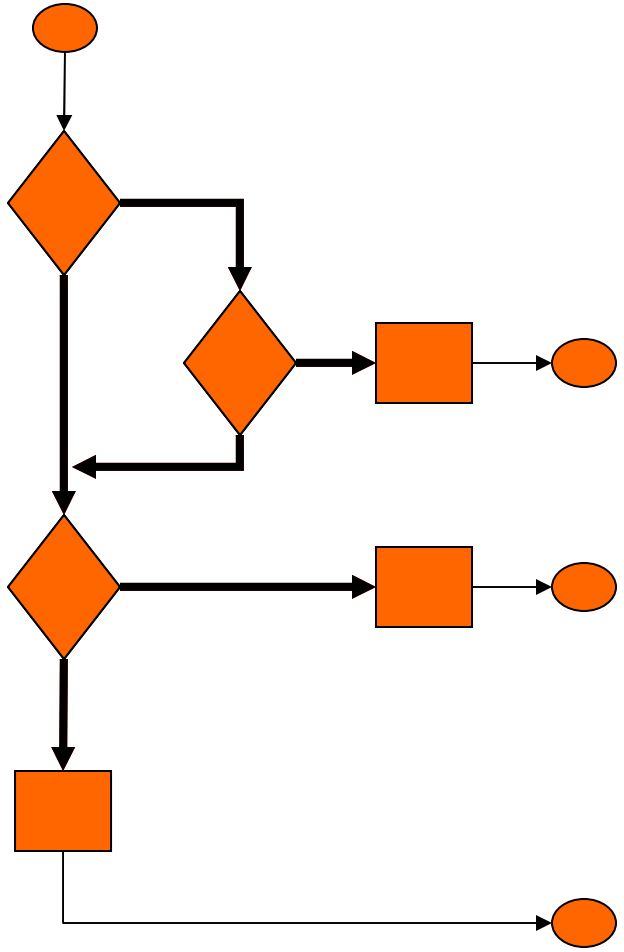
White Box: Statement Coverage

- **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!



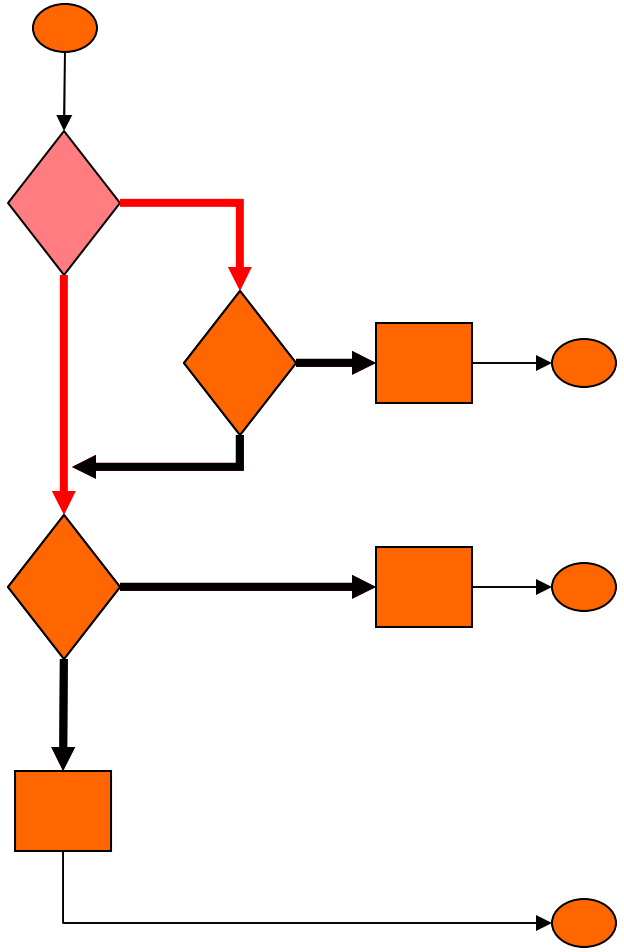
White Box: Branch Coverage

- **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"



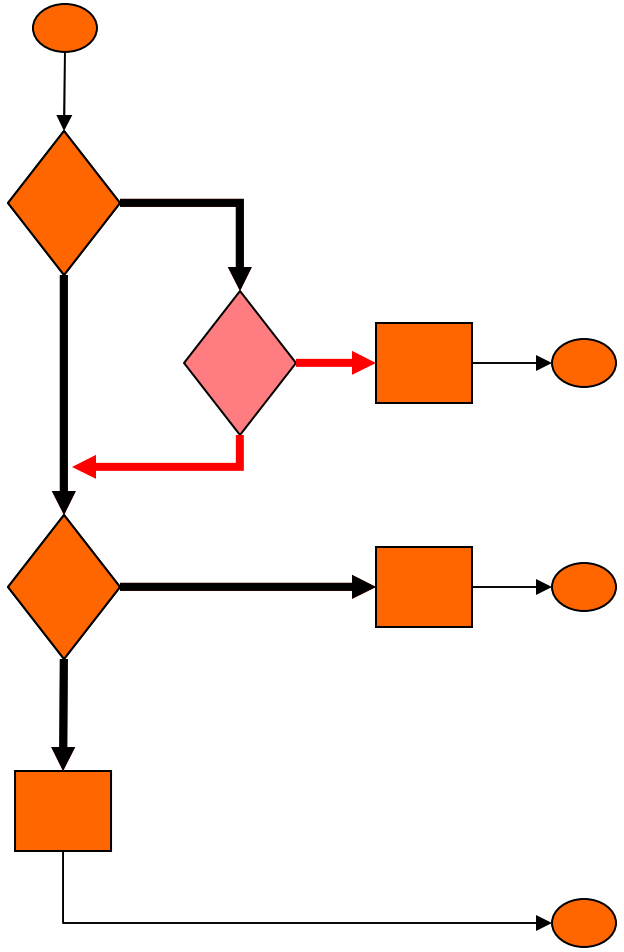
White Box: Branch Coverage

- **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"



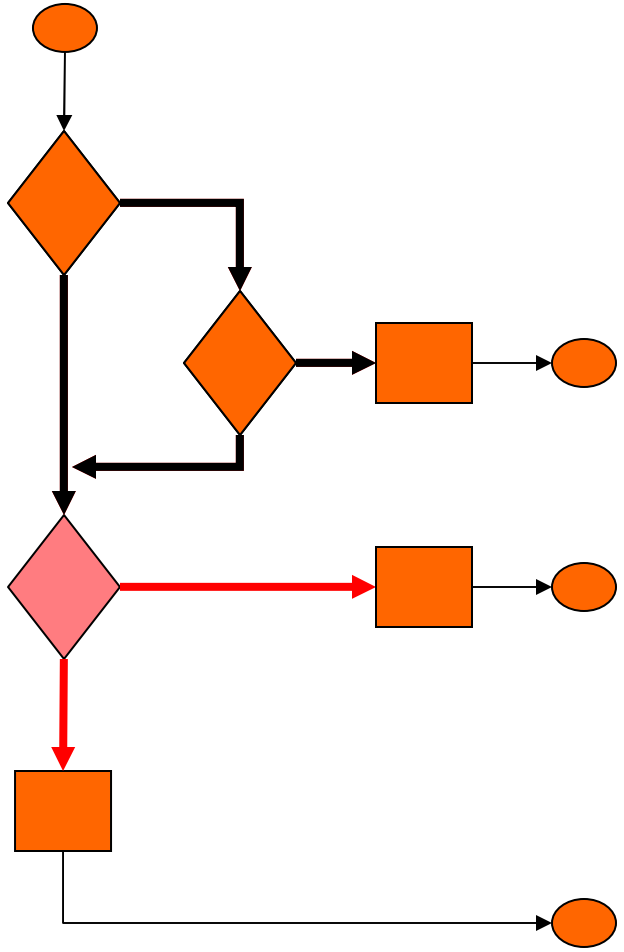
White Box: Branch Coverage

- **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"



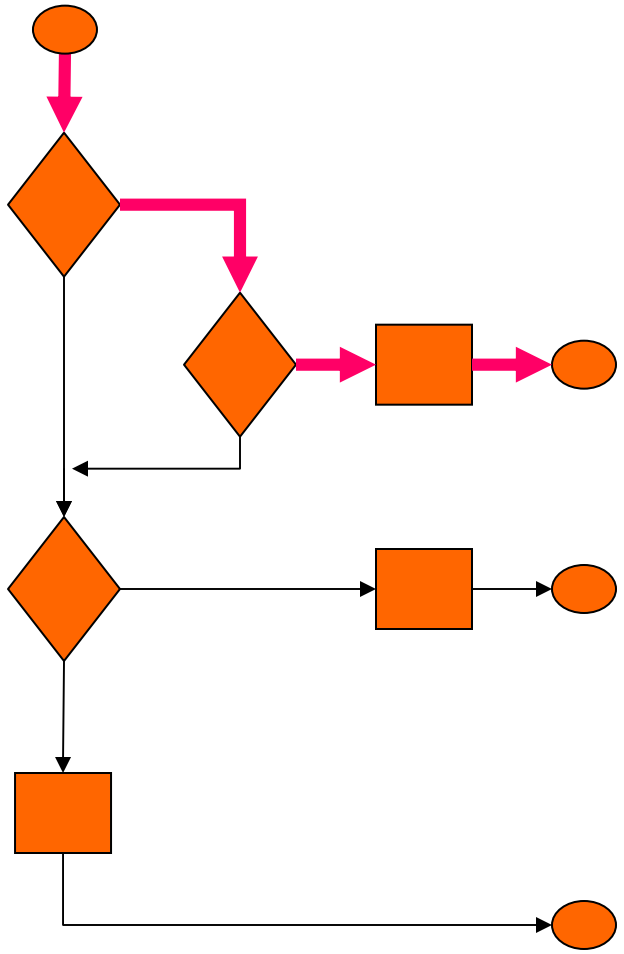
White Box: Branch Coverage

- **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"



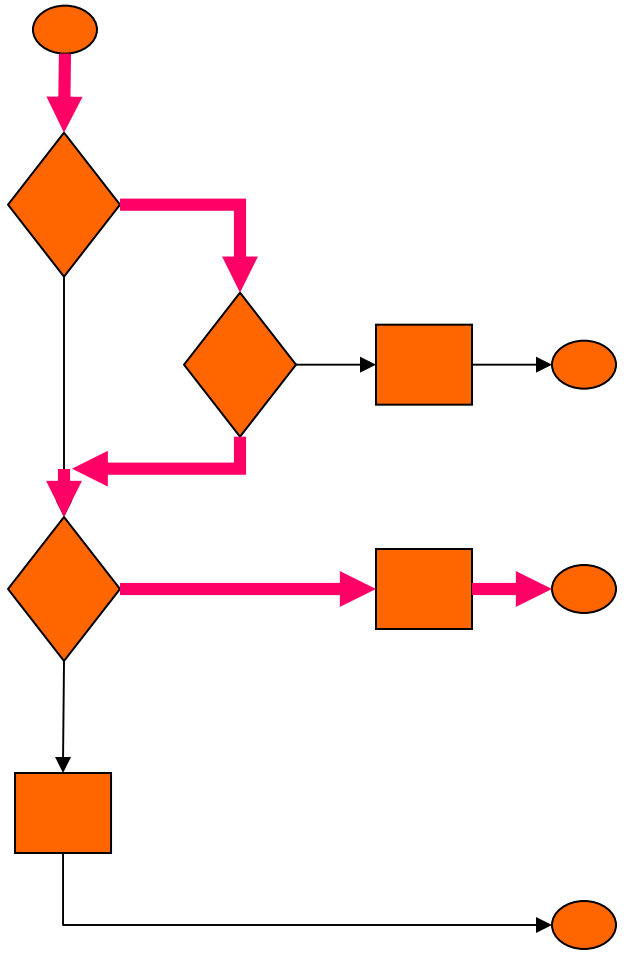
White Box: Path Coverage

- **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



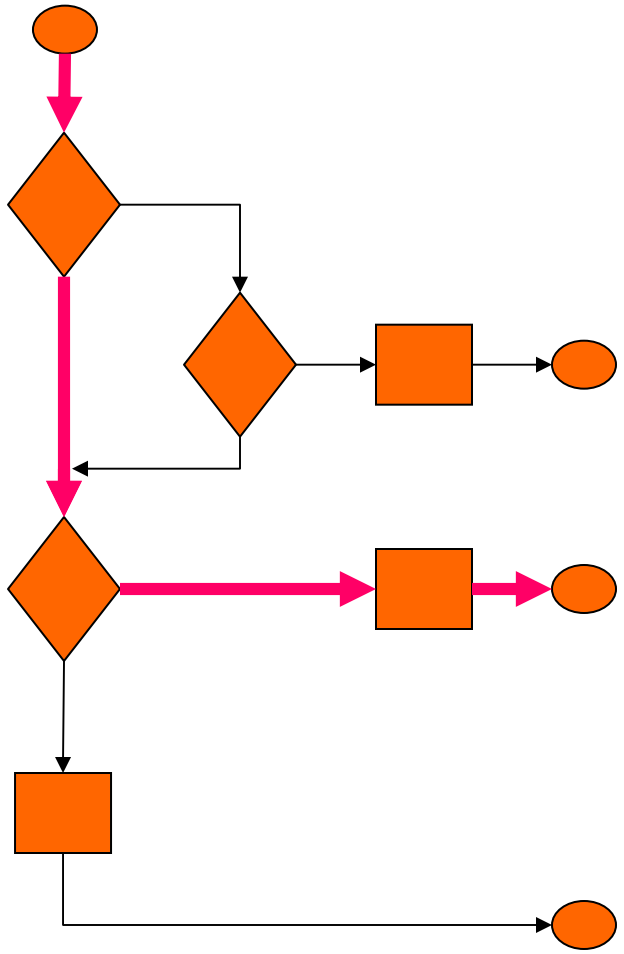
White Box: Path Coverage

- **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



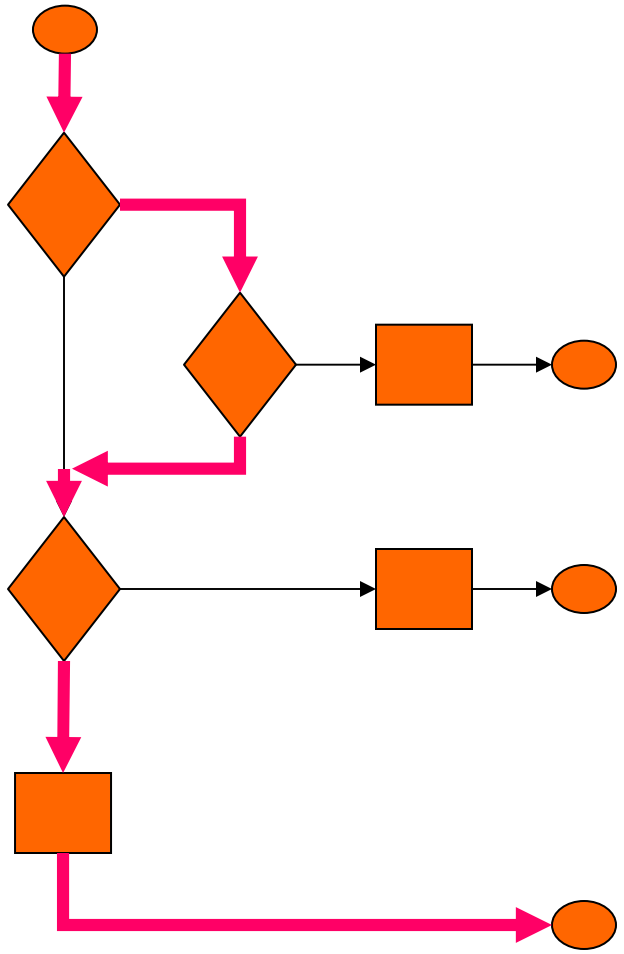
White Box: Path Coverage

- **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



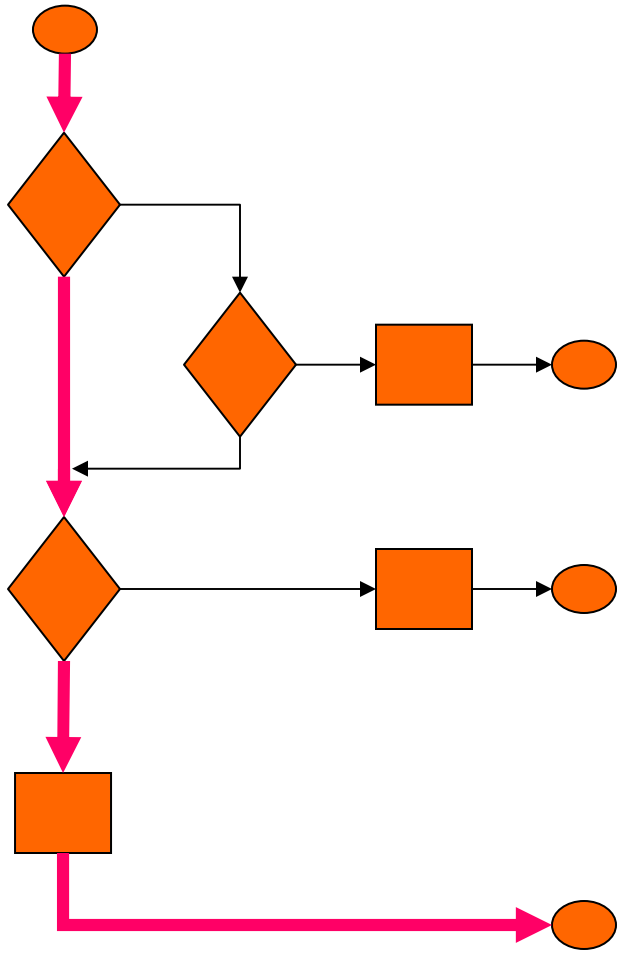
White Box: Path Coverage

- **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



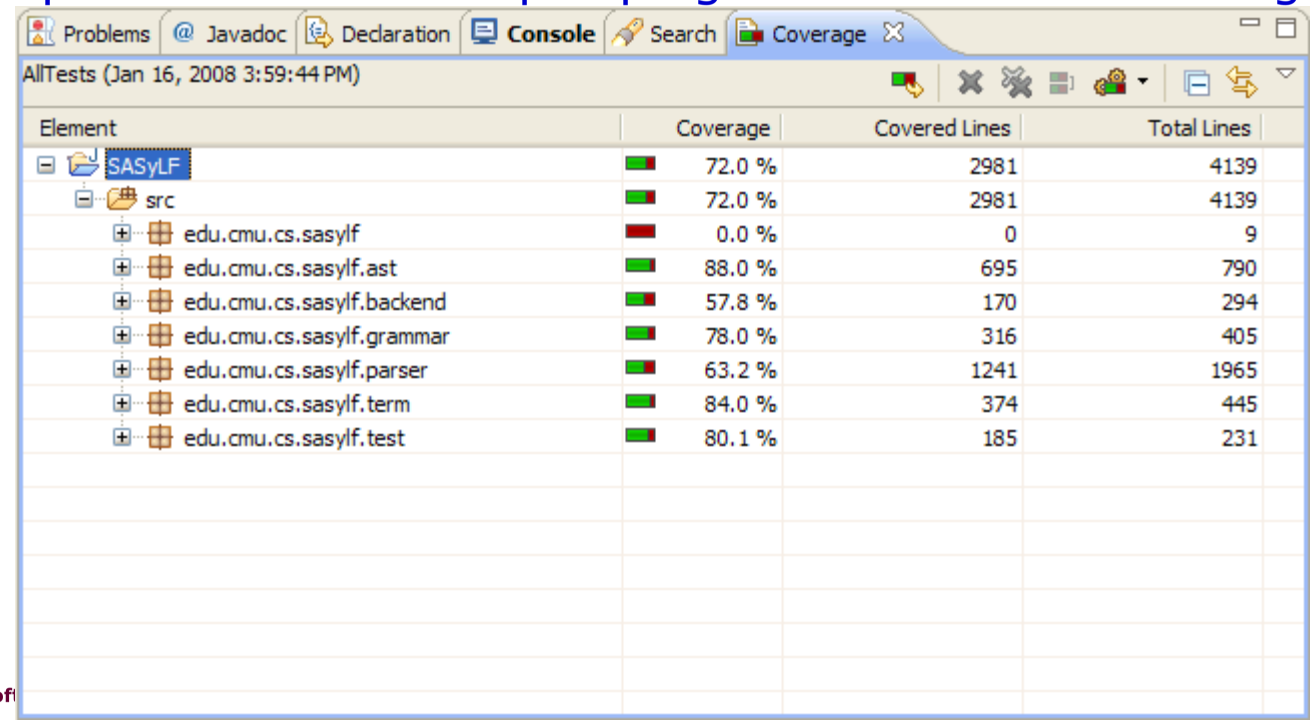
White Box: Path Coverage

- **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

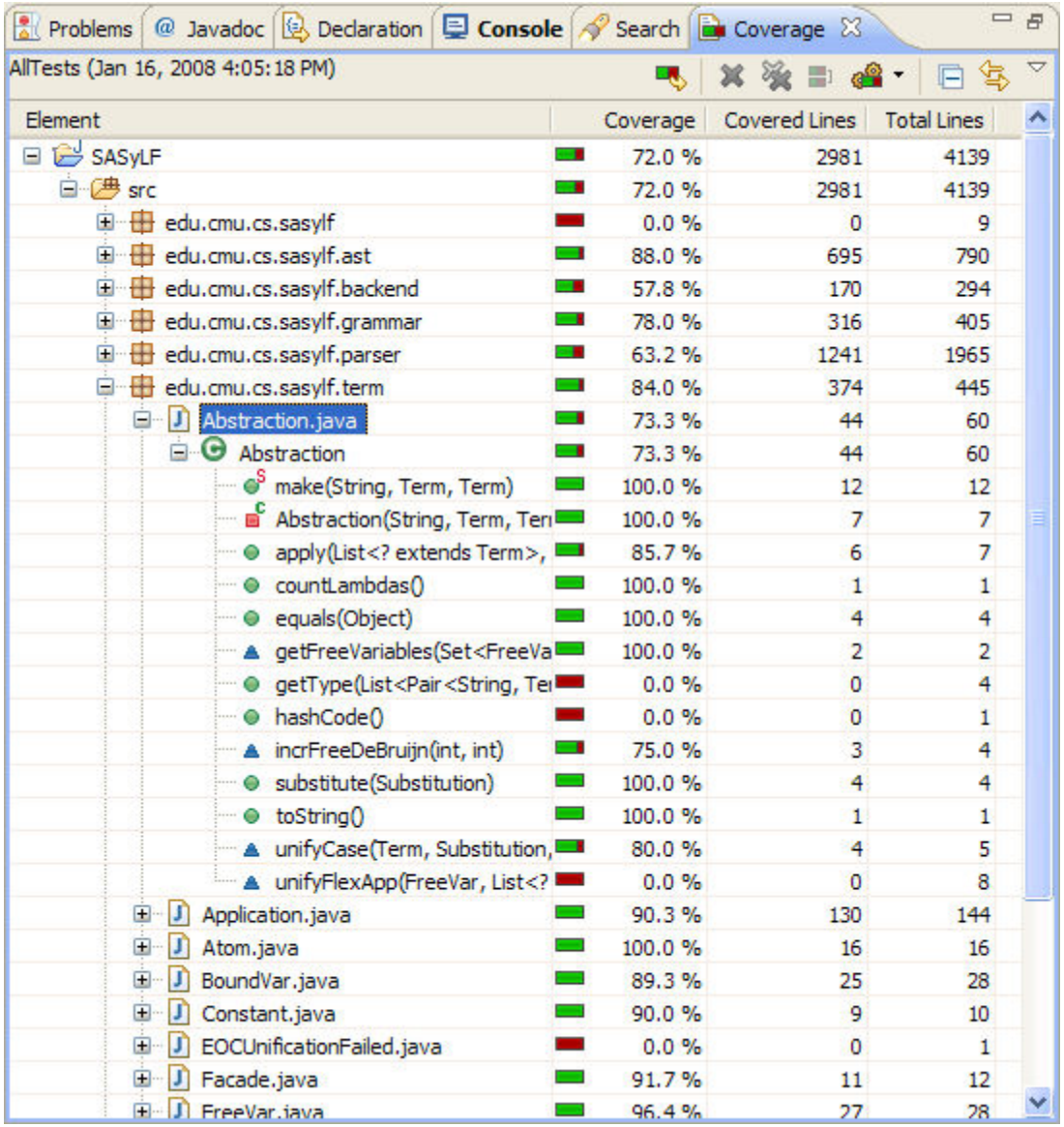
- Coverage assessment tools
  - Track execution of code by test cases
  - Techniques
    - Modified runtime environment (e.g., special JVM)
    - Source code transformation
- Count visits to statements
  - Develop reports with respect to specific coverage criteria
- Example: EclEmma – Eclipse plugin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



Element	Coverage	Covered Lines	Total Lines
SASyLF	72.0 %	2981	4139
src	72.0 %	2981	4139
edu.cmu.cs.sasylf	0.0 %	0	9
edu.cmu.cs.sasylf.ast	88.0 %	695	790
edu.cmu.cs.sasylf.backend	57.8 %	170	294
edu.cmu.cs.sasylf.grammar	78.0 %	316	405
edu.cmu.cs.sasylf.parser	63.2 %	1241	1965
edu.cmu.cs.sasylf.term	84.0 %	374	445
edu.cmu.cs.sasylf.test	80.1 %	185	231

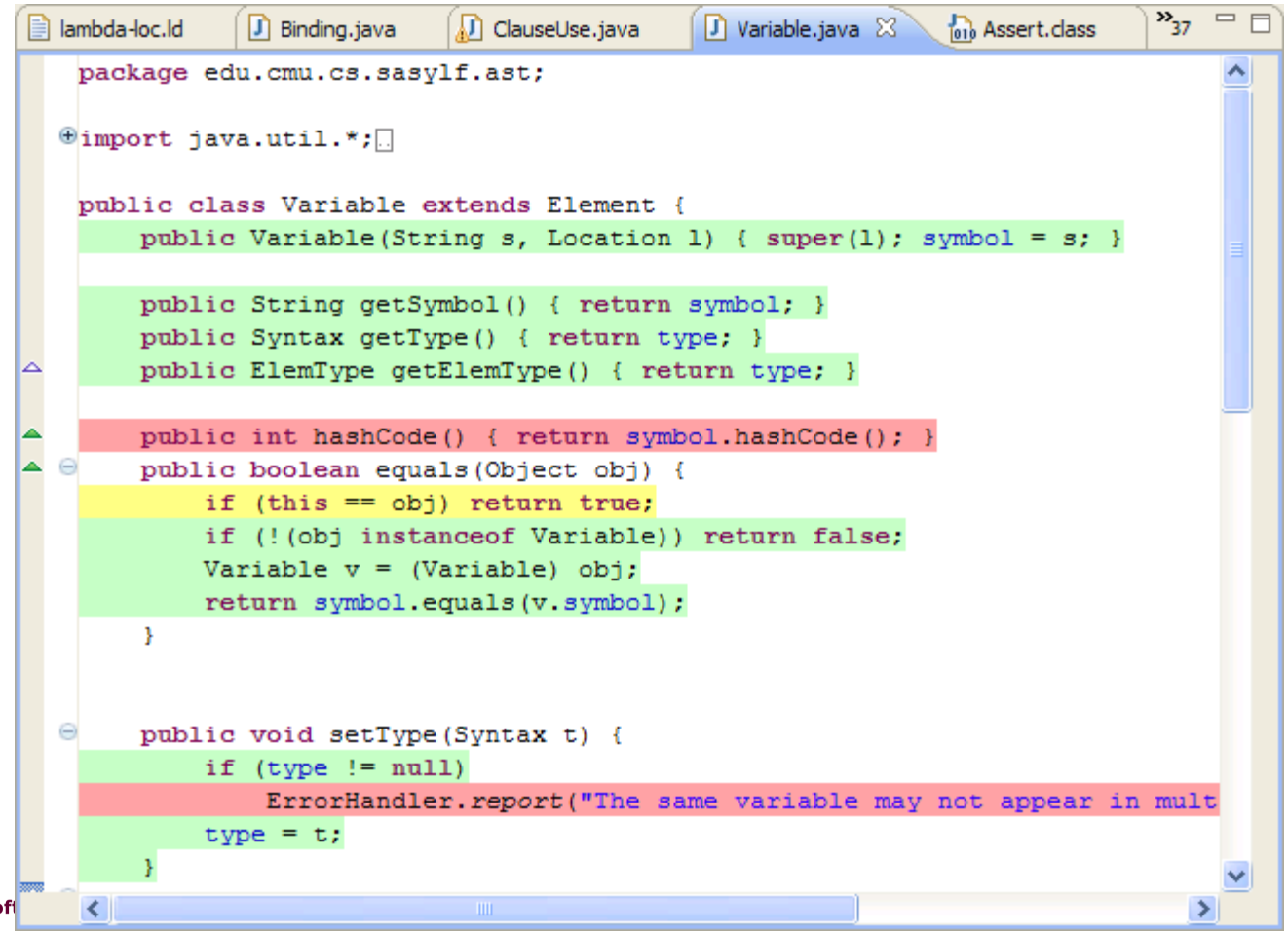
EclEmma in Eclipse

- 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



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?

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*

Testing example

- Equivalence classes?
- Boundary values?
- Robustness tests?
- How to achieve **line coverage**?

```
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;
    }
}
```