

# Testing

© 2009 by Jonathan Aldrich  
Portions © 2007 by William L Scherlis  
used by permission

**No part may be copied or used  
without written permission.**

**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](mailto: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*

# 1. Testing: What and Why

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

# 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. 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*

# Specifications

- Contains
  - Functional behavior
  - Erroneous behavior
  - Quality attributes
- Desirable attributes
  - Complete
    - Does not leave out any desired behavior
  - Minimal
    - Does not require anything that the user does not care about
  - Unambiguous
    - Fully specifies what the system should do in every case the user cares about
  - Consistent
    - Does not have internal contradictions
  - Testable
    - Feasible to objectively evaluate
  - Correct
    - Represents what the end-user(s) need

# Function Specifications

- A function's contract is a statement of the responsibilities of that function, 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
- Contract structure
  - Precondition: the condition the function relies on for correct operation
  - Postcondition: the condition the function establishes after correctly running
- Example:

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



# Function Specifications

- A function's contract is a statement of the responsibilities of that function, 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
- Contract structure
  - Precondition: the condition the function relies on for correct operation
  - Postcondition: the condition the function establishes after correctly running
- Example:

```
/** 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) { ... }
```

# Function Specifications

- A function's contract is a statement of the responsibilities of that function, 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
- Contract structure
  - Precondition: the condition the function relies on for correct operation
  - Postcondition: the condition the function establishes after correctly running
- (Functional) correctness with respect to the specification
  - If the client of a function fulfills the function's precondition, the function will execute to completion and when it terminates, the postcondition will be fulfilled
- What does the implementation have to fulfill if the client violates the precondition?

# Function Specifications

- A function's contract is a statement of the responsibilities of that function, 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
- Contract structure
  - Precondition: the condition the function relies on for correct operation
  - Postcondition: the condition the function establishes after correctly running
- (Functional) correctness with respect to the specification
  - If the client of a function fulfills the function's precondition, the function will execute to completion and when it terminates, the postcondition will be fulfilled
- What does the implementation have to fulfill if the client violates the precondition?
  - A: Nothing. It can do anything at all.

## Quick Quiz

Assume the specification for sum given in the lecture slides:

**requires** array != null && len >= 0 && array.length == len

**ensures** \result == (\sum int j; 0 <= j && j < len; array[j])

Assume the following input and outputs for sum, where a 3 element array is written as [1, 2, 3]. For which of the inputs and outputs is the implementation of sum correct according to the specification given?

- Input: array = [1, 2, 3, 4], len = 4  
Output: 10
- Input: array = [0, 0, 3, -7], len = 4  
Output: *none (the program does not terminate)*
- Input: array = [1, 2, 3, 4], len = 3  
Output: 7
- Input: array = [1, 2, -3, 4], len = 4  
Output: 7

# Erroneous Behavior Specifications

- A function can do anything at all if precondition is violated, BUT...
  - we may want the system to function even if one part fails
  - we may want to easily identify our mistakes
- Exceptional case specifications
  - Precondition: condition describing the input that leads to an error
  - Postcondition: condition established by the function under that erroneous input
- Example (BitSet.toArray() in JML)

```
/*@ public normal_behavior
  @ requires a != null;
  @ requires (\forall Object o; containsObject(o);
             \typeof(o) <: \elemtype(\typeof(a)));
  @ also
  @ public exceptional_behavior
  @ requires a == null;
  @ signals_only NullPointerException ;
  @ also
  @ public exceptional_behavior
  @ requires a != null;
  @ requires !(\forall Object o; containsObject(o);
              \typeof(o) <: \elemtype(\typeof(a)));
  @ signals_only ArrayStoreException ;
  @*/
Object[] toArray(Object[] a) throws NullPointerException, ArrayStoreException;
```

## Example Java I/O Library Specification (abridged)

public int **read**(byte[] b, int off, int len) throws IOException

- Reads up to len bytes of data from the input stream into an array of bytes. An attempt is made to read as many as len bytes, but a smaller number may be read. The number of bytes actually read is returned as an integer. This method blocks until input data is available, end of file is detected, or an exception is thrown.
- If len is zero, then no bytes are read and 0 is returned; otherwise, there is an attempt to read at least one byte. If no byte is available because the stream is at end of file, the value -1 is returned; otherwise, at least one byte is read and stored into b.
- The first byte read is stored into element b[off], the next one into b[off+1], and so on. The number of bytes read is, at most, equal to len. Let k be the number of bytes actually read; these bytes will be stored in elements b[off] through b[off+k-1], leaving elements b[off+k] through b[off+len-1] unaffected.
- In every case, elements b[0] through b[off] and elements b[off+len] through b[b.length-1] are unaffected.

### • **Throws:**

- IOException - If the first byte cannot be read for any reason other than end of file, or if the input stream has been closed, or if some other I/O error occurs.
- NullPointerException - If b is null.
- IndexOutOfBoundsException - If off is negative, len is negative, or len is greater than b.length - off

# Example Java I/O Library Specification (abridged)

public int **read**(byte[] b, int off, int len) throws IOException

- Reads up to len bytes of data from the input stream. An attempt is made to read len bytes. The number of bytes actually read may be less than len. The number of bytes read is returned as an integer. This method blocks until some data is available, or EOF is detected, or an exception is thrown.
  - If len is zero, then no bytes are read. If len is greater than zero, there is an attempt to read at least one byte. If the stream is at end of file, the value returned is -1. If one byte is read and stored into b, then the next byte is read and stored into b[off+1], and so on. The number of bytes read is returned. Let k be the number of bytes read. The bytes are stored in elements b[off] through b[off+k-1].
  - The first byte read is stored into element b[off], the next into b[off+1], and so on. The number of bytes read is returned. Let k be the number of bytes read. The bytes are stored in elements b[off] through b[off+k-1].
  - In every case, elements b[0] through b[off-1] and elements b[off+len] through b[b.length-1] are unaffected.
- Specification of return
    - len=0 → return 0
    - len>0 && eof → return -1
    - len>0 && !eof → return >0
  - Exactly where the data is stored
  - What parts of the array are **not** affected
- **Throws:**
    - IOException - If the first byte cannot be read from the input stream, or if the input stream has reached the end of file, or if the input stream has an error.
    - NullPointerException - If b is null.
    - IndexOutOfBoundsException - If off is negative, or off+len is greater than b.length - off
  - Multiple error cases, each with a precondition
  - Includes “runtime exceptions” not in throws clause

# Quality Attribute Specifications: Discussion

- How would you specify...
  - Availability?
  - Modifiability?
  - Performance?
  - Security?
  - Usability?



# 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*

# 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

# 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

# Black Box: Equivalence Class / Partition Testing

- Equivalence classes
  - A partition of a set
    - Usually the input domain of the program
  - Based on some equivalence relation
    - Intuition: all inputs in an equivalence class will fail or succeed in the same way

# Finding Equivalence Classes

- Cases in the specification
  - You may not have a spec – but still helps to think in this way
  - Remember that the spec may not be complete!
- One class per code path
  - Really white-box testing – boundary is fuzzy
  - Even with black box it can be useful to guess at the code structure
  - You may have an abstract algorithm to use
- Risk-based
  - Consider a possible error as a risk
  - Given an error, what classes of input could produce that error?
- Guideline – avoid writing many similar test cases
  - Suggests they are all from the same equivalence class

# Equivalence Class Hueristics

- Invalid inputs
- Ranges of numbers
- Membership in a group
- Equivalent outputs
  - Can you force the program to output an invalid or overflow value?
- Error messages
- Equivalent operating environments

# Boundary Value Testing

- What test case to choose from an equivalence class?
  - Which is most useful, i.e. likely to fail?
- Heuristic – boundary values
  - Extreme or unique cases at or around “boundaries”
    - Boundary of precondition – black box
    - Boundary of program decision point – white box
    - *Examples*: zero-length inputs, very long inputs, null references, etc.
  - Will usually find errors that are present in any other member of the equivalence class, but may find off-by-one errors as well

# Combination Testing

- Some errors might be triggered only if two or more variables are at boundary values
- Test combinations of boundary values
  - Combinations of valid input
  - One invalid input at a time
    - In many cases no added value for multiple invalid inputs
- Subtlety required
  - What are the boundary cases for an application that deals with months and days?



# Robustness Testing

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

**a[mid]**

**What if the array reference a is null?**

# Equivalence, Boundary and Robustness Example

- Program Specification

- Given numbers  $a$ ,  $b$ , and  $c$ , return the roots of the quadratic polynomial  $ax^2 + bx + c$ . Recall that the roots of a quadratic equation are given by:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

- Equivalence classes?

- Robustness test cases?

# 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?

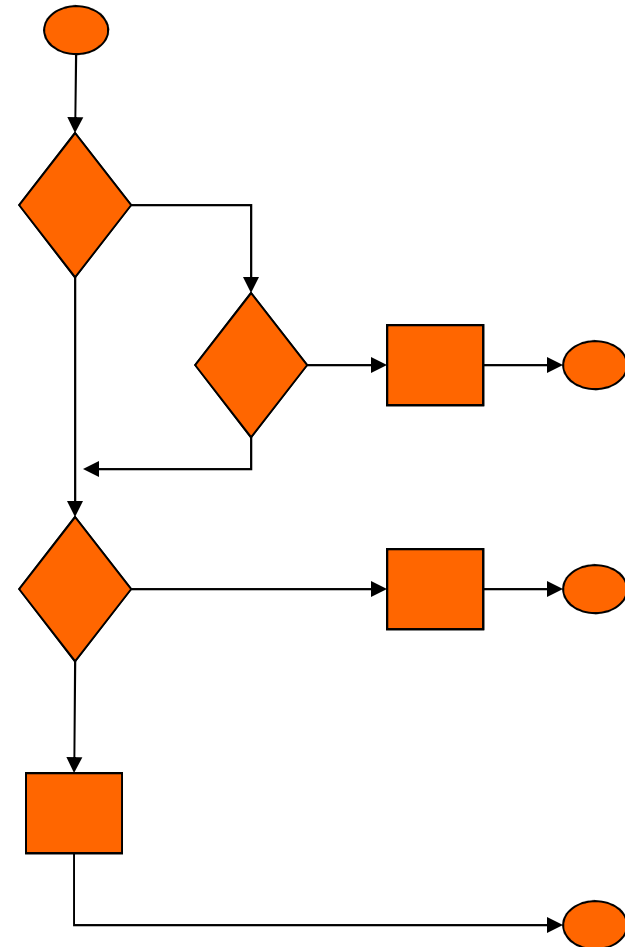
# Test Coverage

- 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



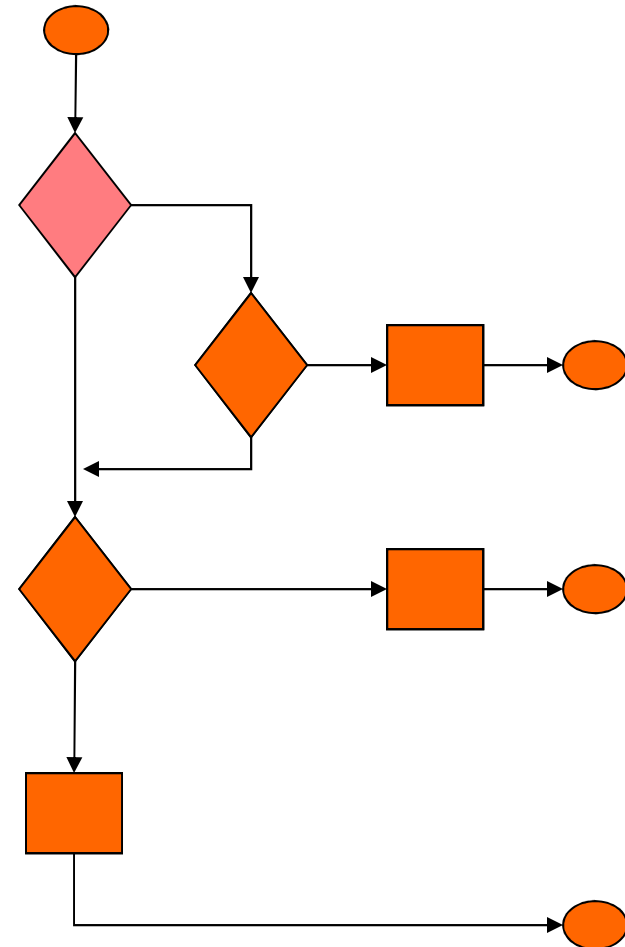
# White Box: Statement Coverage

- **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!



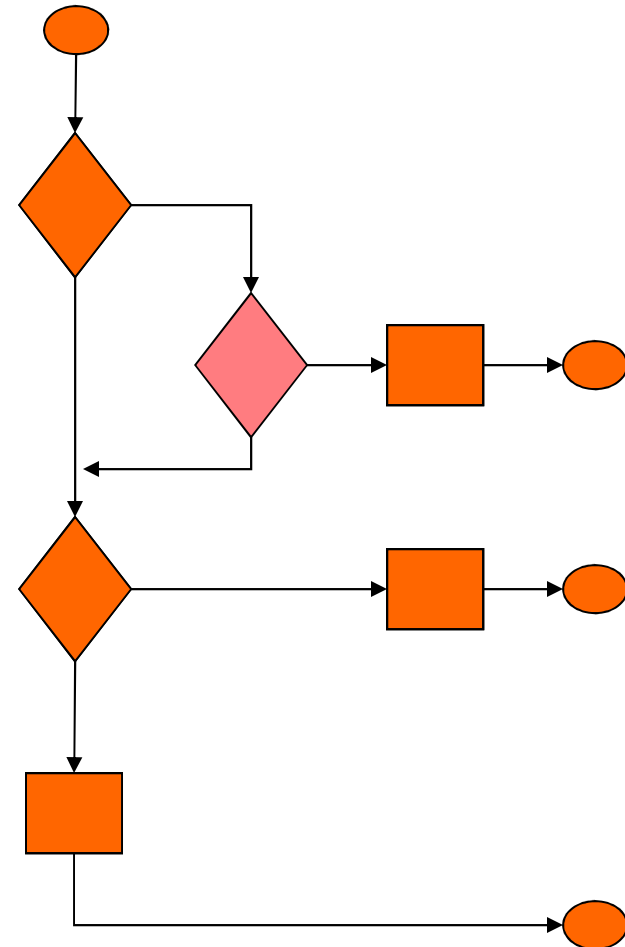
# White Box: Statement Coverage

- **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!



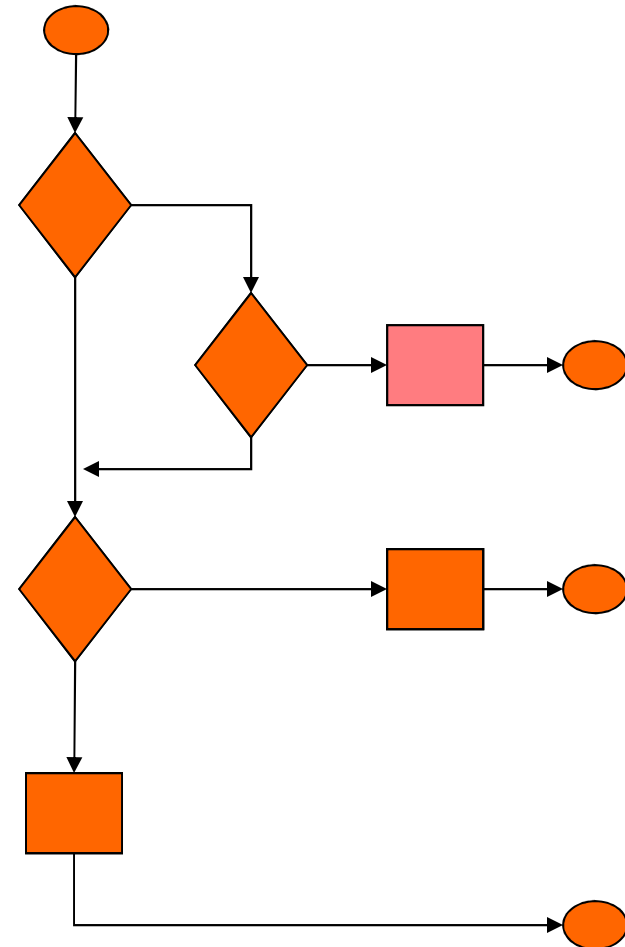
# White Box: Statement Coverage

- **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!



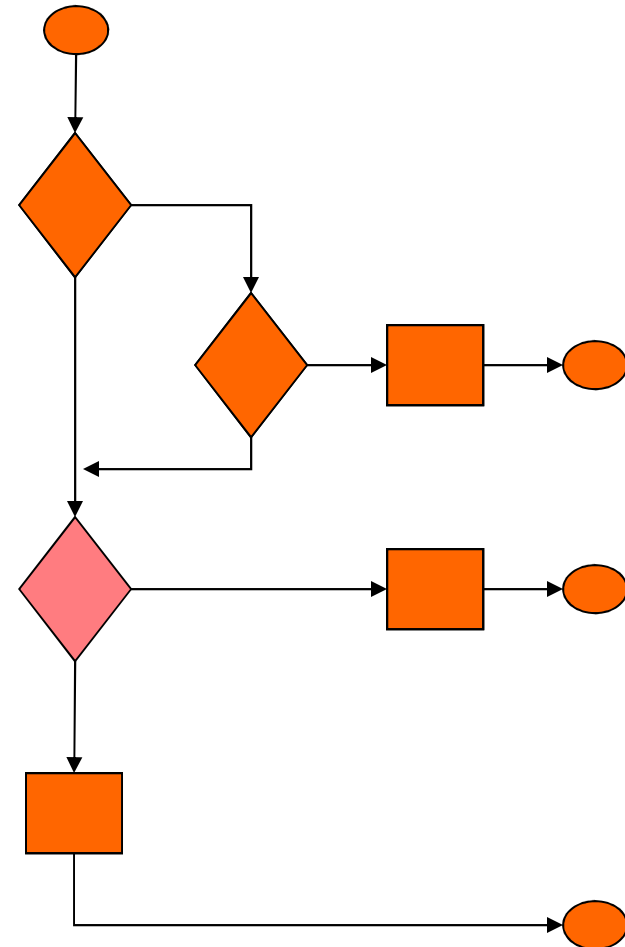
# White Box: Statement Coverage

- **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!



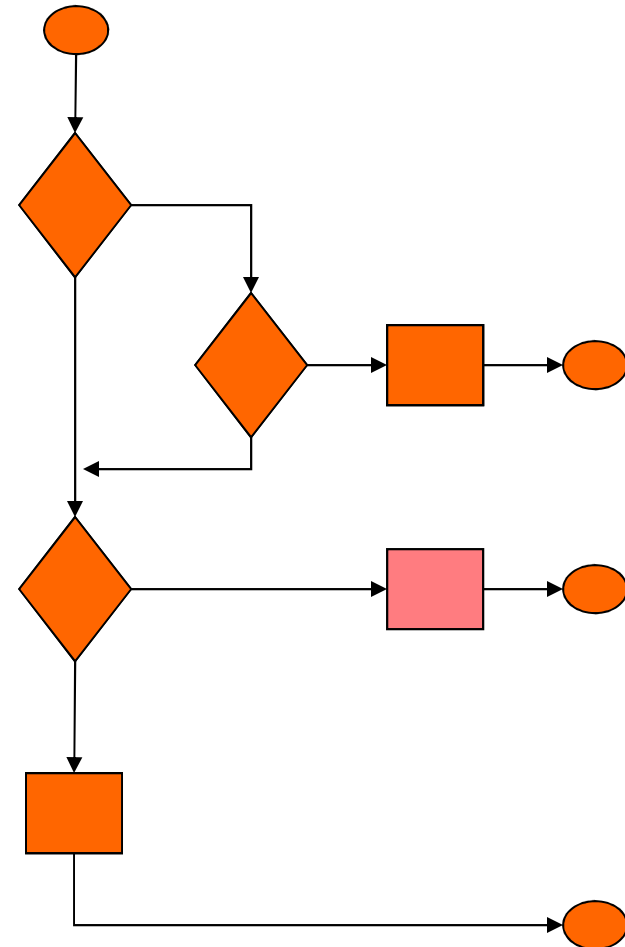
# White Box: Statement Coverage

- **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!



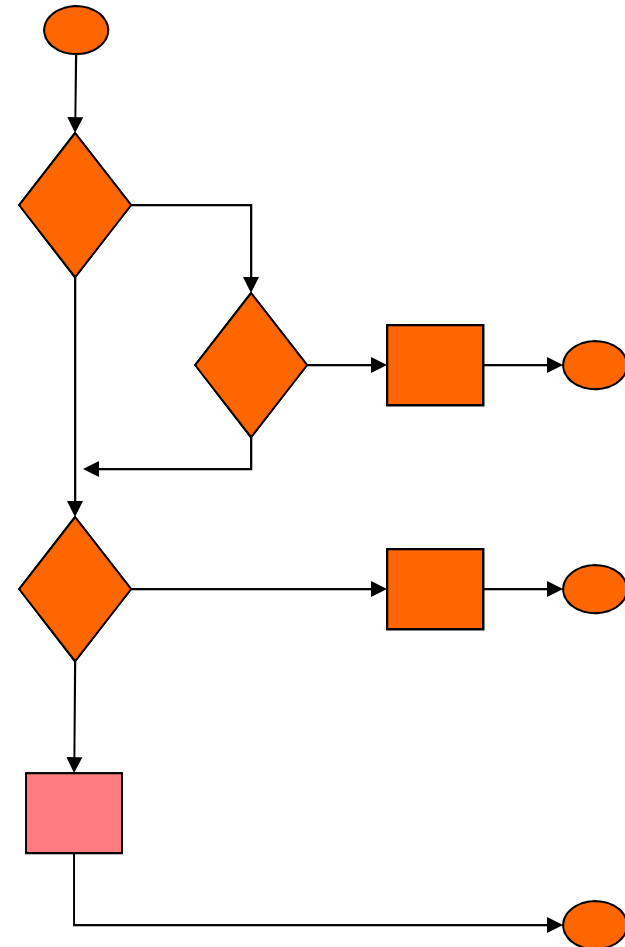
# White Box: Statement Coverage

- **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**

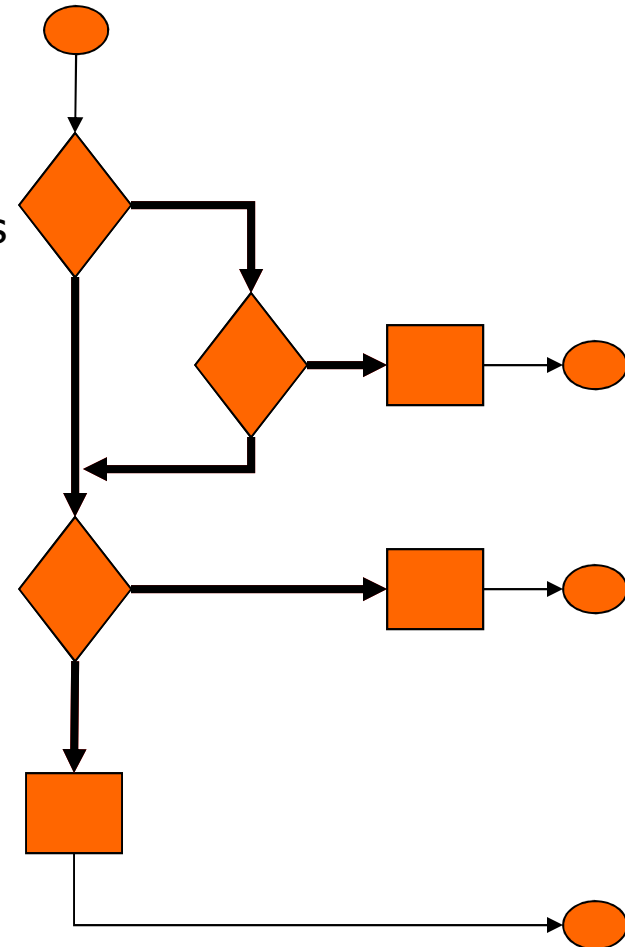
- What portion of condition branches are covered by test cases?
- *Or:* What portion of relational expressions and values are covered by test cases?
  - Condition testing (Tai)
- **Multicondition coverage** – all boolean combinations of tests are covered

- **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**

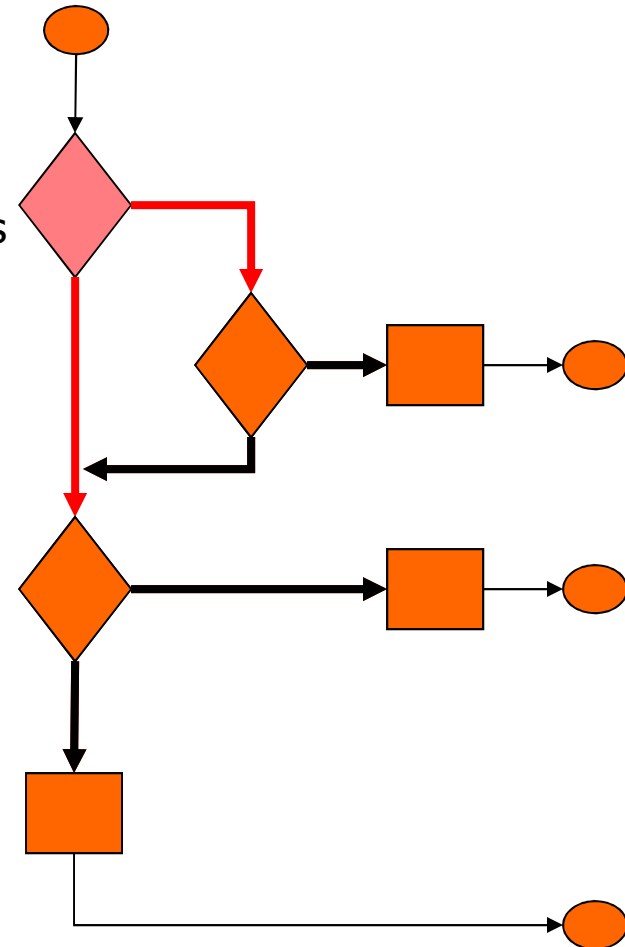
- What portion of condition branches are covered by test cases?
- *Or:* What portion of relational expressions and values are covered by test cases?
  - Condition testing (Tai)
- **Multicondition coverage** – all boolean combinations of tests are covered

- **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**

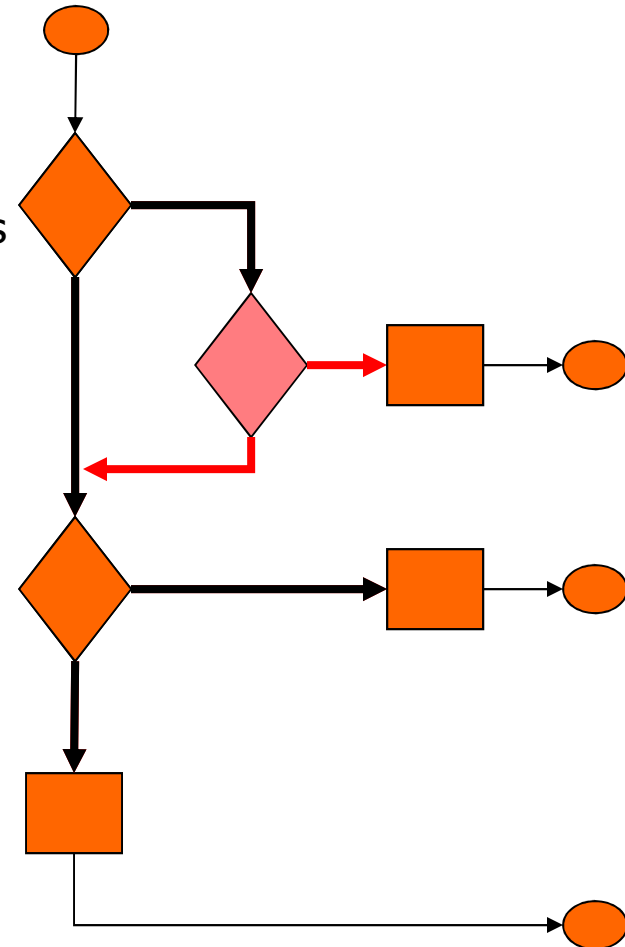
- What portion of condition branches are covered by test cases?
- *Or:* What portion of relational expressions and values are covered by test cases?
  - Condition testing (Tai)
- **Multicondition coverage** – all boolean combinations of tests are covered

- **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**

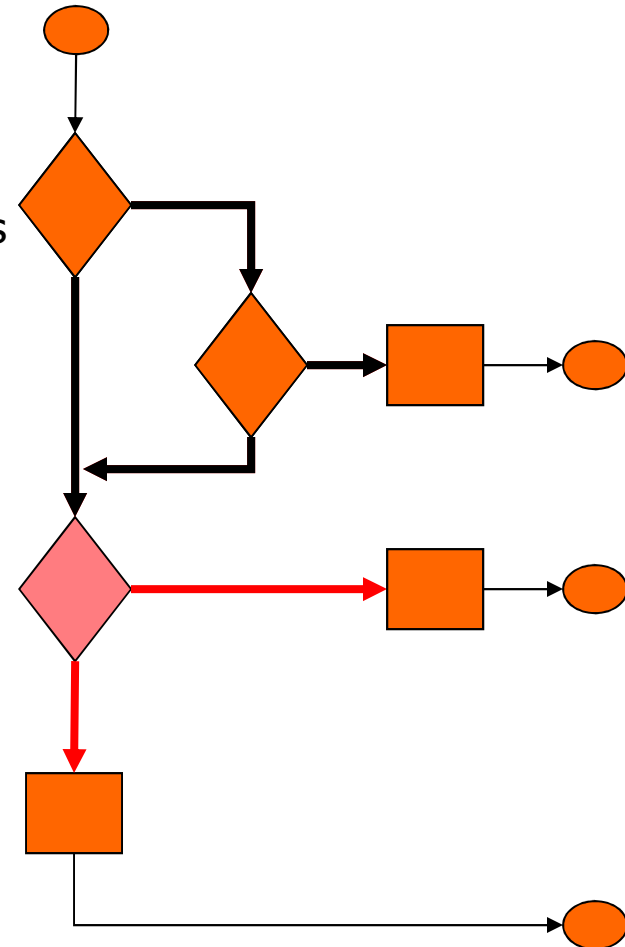
- What portion of condition branches are covered by test cases?
- *Or:* What portion of relational expressions and values are covered by test cases?
  - Condition testing (Tai)
- **Multicondition coverage** – all boolean combinations of tests are covered

- **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**

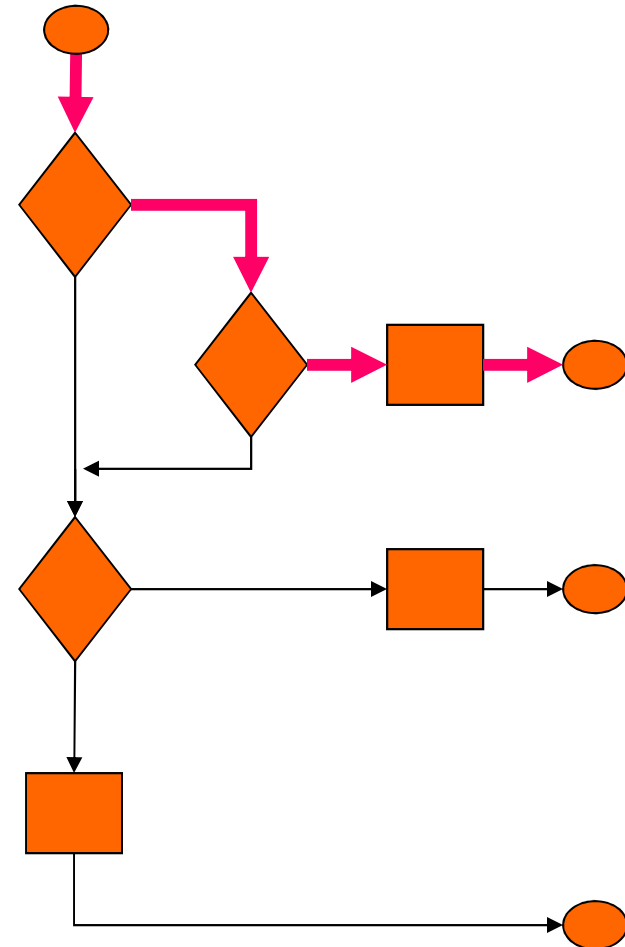
- What portion of all possible paths through the program are covered by tests?
- Loop testing: Consider representative and edge cases:
  - Zero, one, two iterations
  - If there is a bound  $n$ :  $n-1$ ,  $n$ ,  $n+1$  iterations
  - Nested loops/conditionals from inside out

- **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**

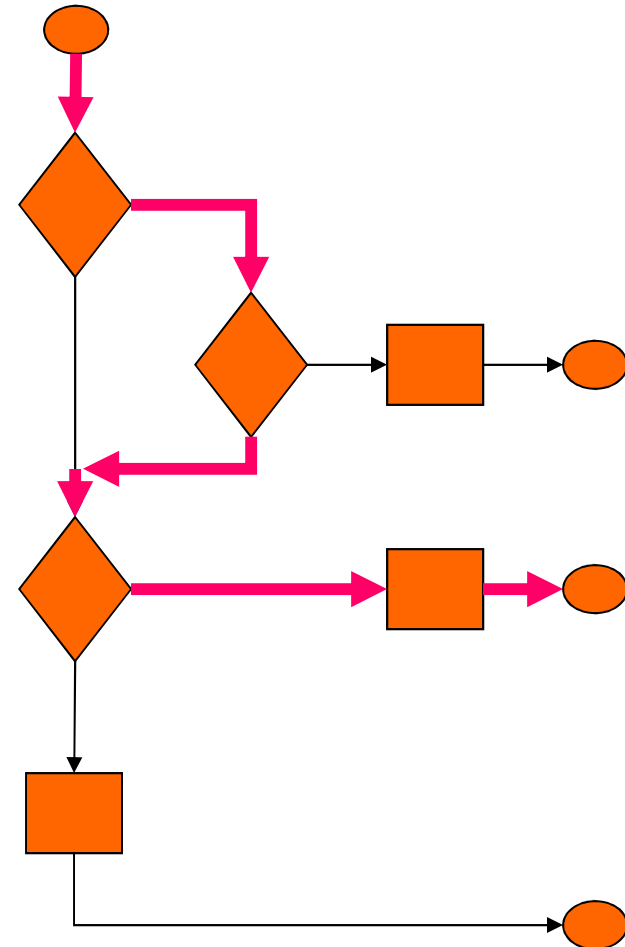
- What portion of all possible paths through the program are covered by tests?
- Loop testing: Consider representative and edge cases:
  - Zero, one, two iterations
  - If there is a bound  $n$ :  $n-1$ ,  $n$ ,  $n+1$  iterations
  - Nested loops/conditionals from inside out

- **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**

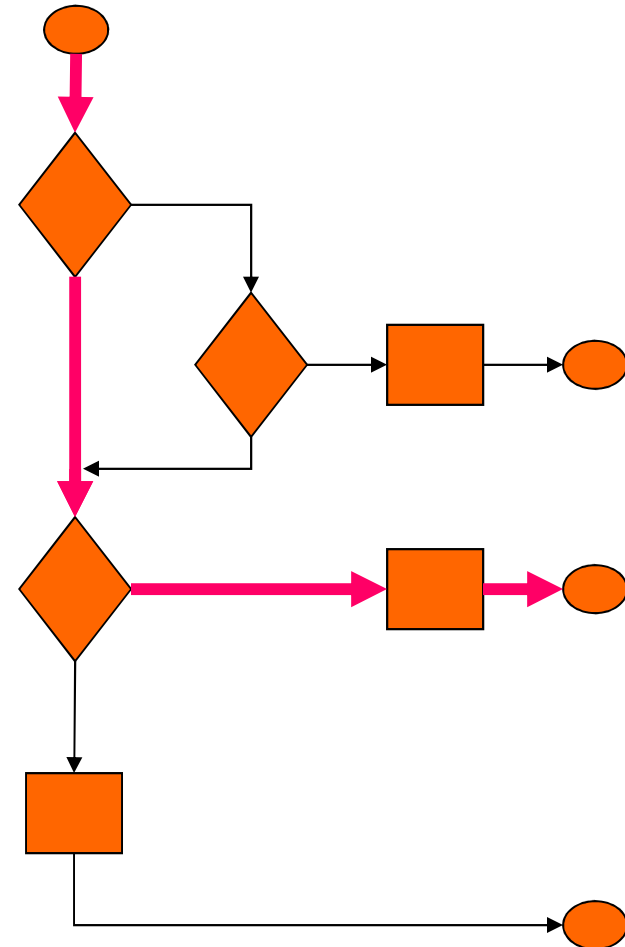
- What portion of all possible paths through the program are covered by tests?
- Loop testing: Consider representative and edge cases:
  - Zero, one, two iterations
  - If there is a bound  $n$ :  $n-1$ ,  $n$ ,  $n+1$  iterations
  - Nested loops/conditionals from inside out

- **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**

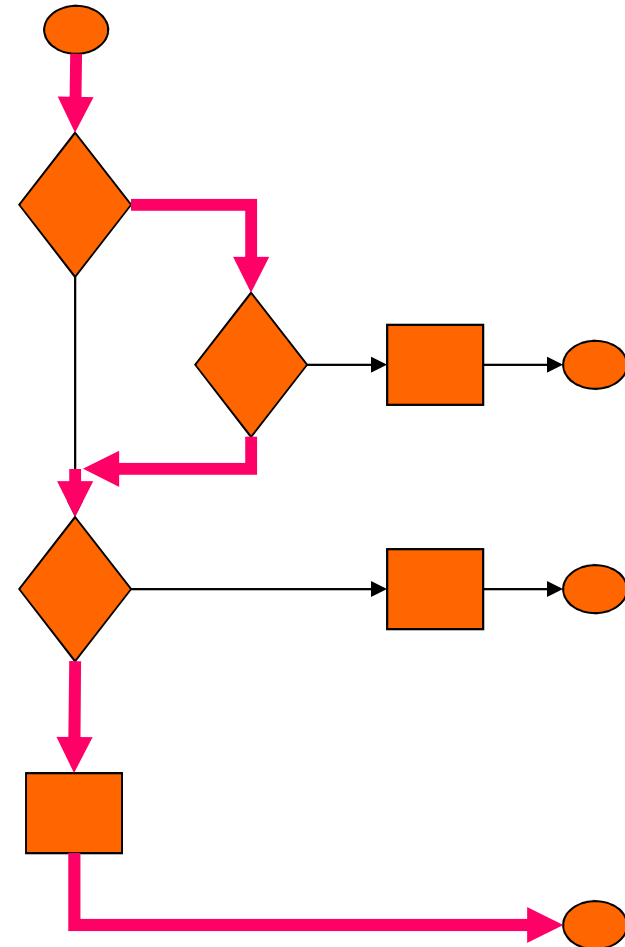
- What portion of all possible paths through the program are covered by tests?
- Loop testing: Consider representative and edge cases:
  - Zero, one, two iterations
  - If there is a bound  $n$ :  $n-1$ ,  $n$ ,  $n+1$  iterations
  - Nested loops/conditionals from inside out

- **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**

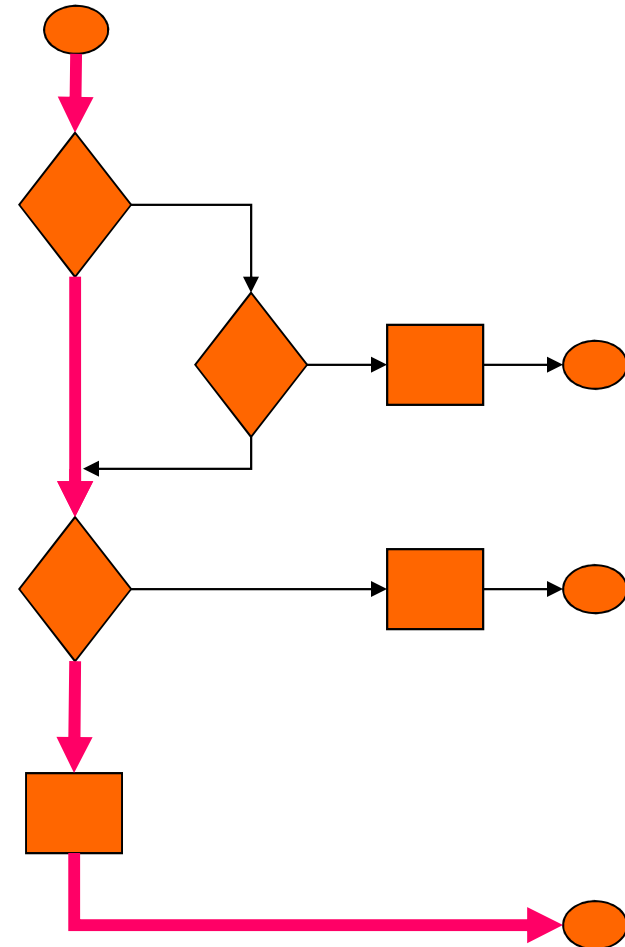
- What portion of all possible paths through the program are covered by tests?
- Loop testing: Consider representative and edge cases:
  - Zero, one, two iterations
  - If there is a bound  $n$ :  $n-1$ ,  $n$ ,  $n+1$  iterations
  - Nested loops/conditionals from inside out

- **Advantages**

- Better coverage of logical flows

- **Disadvantages**

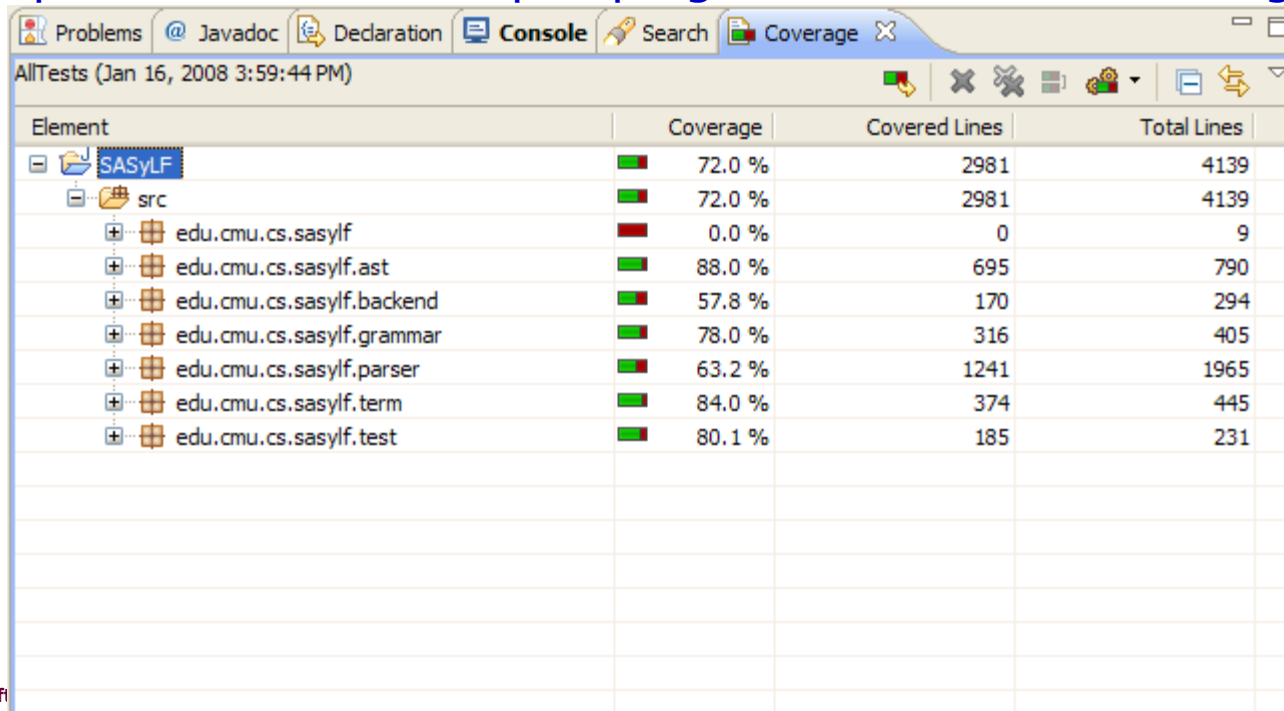
- Not all paths are possible, or necessary
  - What are the *significant* paths?
- Combinatorial explosion in cases unless careful choices are made
  - E.g., sequence of  $n$  **if** tests can yield up to  $2^n$  possible paths
- Assumption that program structure is basically sound














## 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: EcEmma – 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

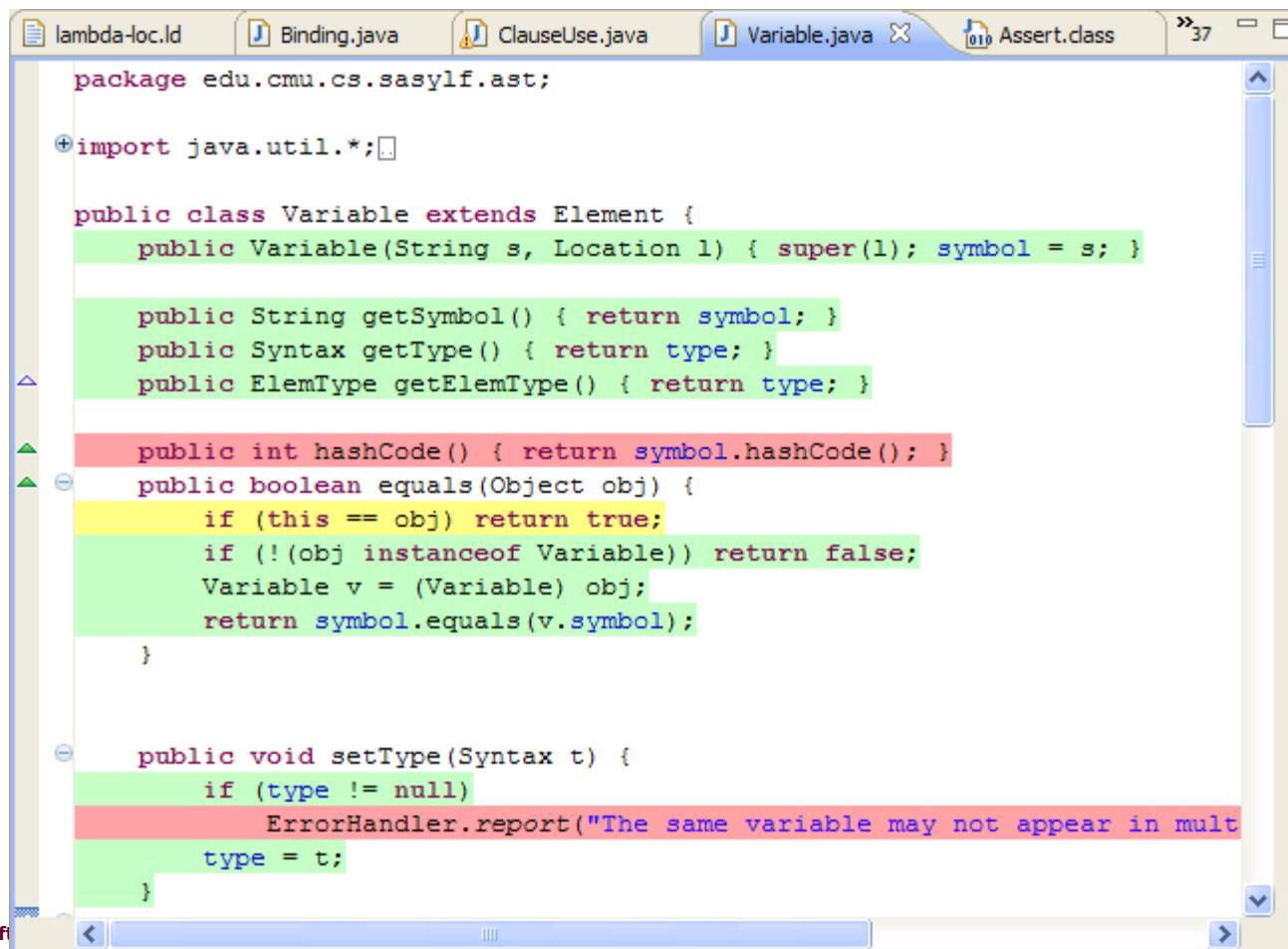
# EclEmma in Eclipse

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

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
Abstraction.java	73.3 %	44	60
Abstraction	73.3 %	44	60
make(String, Term, Term)	100.0 %	12	12
Abstraction(String, Term, Term)	100.0 %	7	7
apply(List<? extends Term>, Substitution)	85.7 %	6	7
countLambdas()	100.0 %	1	1
equals(Object)	100.0 %	4	4
getFreeVariables(Set<FreeVar>)	100.0 %	2	2
getType(List<Pair<String, Term>>)	0.0 %	0	4
hashCode()	0.0 %	0	1
incrFreeDeBruijn(int, int)	75.0 %	3	4
substitute(Substitution)	100.0 %	4	4
toString()	100.0 %	1	1
unifyCase(Term, Substitution, Substitution)	80.0 %	4	5
unifyFlexApp(FreeVar, List<? extends Term>)	0.0 %	0	8
Application.java	90.3 %	130	144
Atom.java	100.0 %	16	16
BoundVar.java	89.3 %	25	28
Constant.java	90.0 %	9	10
EOCUnificationFailed.java	0.0 %	0	1
Facade.java	91.7 %	11	12
FreeVar.java	96.4 %	27	28

# Clover in Eclipse

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



```
package edu.cmu.cs.sasylf.ast;

import java.util.*;

public class Variable extends Element {
    public Variable(String s, Location l) { super(l); symbol = s; }

    public String getSymbol() { return symbol; }
    public Syntax getType() { return type; }
    public ElemType getElemType() { return type; }

    public int hashCode() { return symbol.hashCode(); }
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (!(obj instanceof Variable)) return false;
        Variable v = (Variable) obj;
        return symbol.equals(v.symbol);
    }

    public void setType(Syntax t) {
        if (type != null)
            ErrorHandler.report("The same variable may not appear in mult
        type = t;
    }
}
```

## 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;  
    }  
}
```

# 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*



## 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  $\sim$  % 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  $n_1=10$  defects
  - Inspector B finds  $n_2=8$  defects
  - $m = 5$  defects found by both A and B
  - $N$  is the (unknown) number of defects in the software
- **Lincoln-Petersen analysis** [source: Wikipedia]
  - Consider just the 10 (total) defects found by A
  - Inspector B found 5 of these 10 defects
  - Therefore the probability that inspector B finds a given defect is 5/10 or 50%
  - So, inspector B should have found 50% of the  $N$  defects in the software, so

$$N = n_1 * n_2 / m = 10 * 8 / 5 = 16 \text{ defects}$$

- **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...

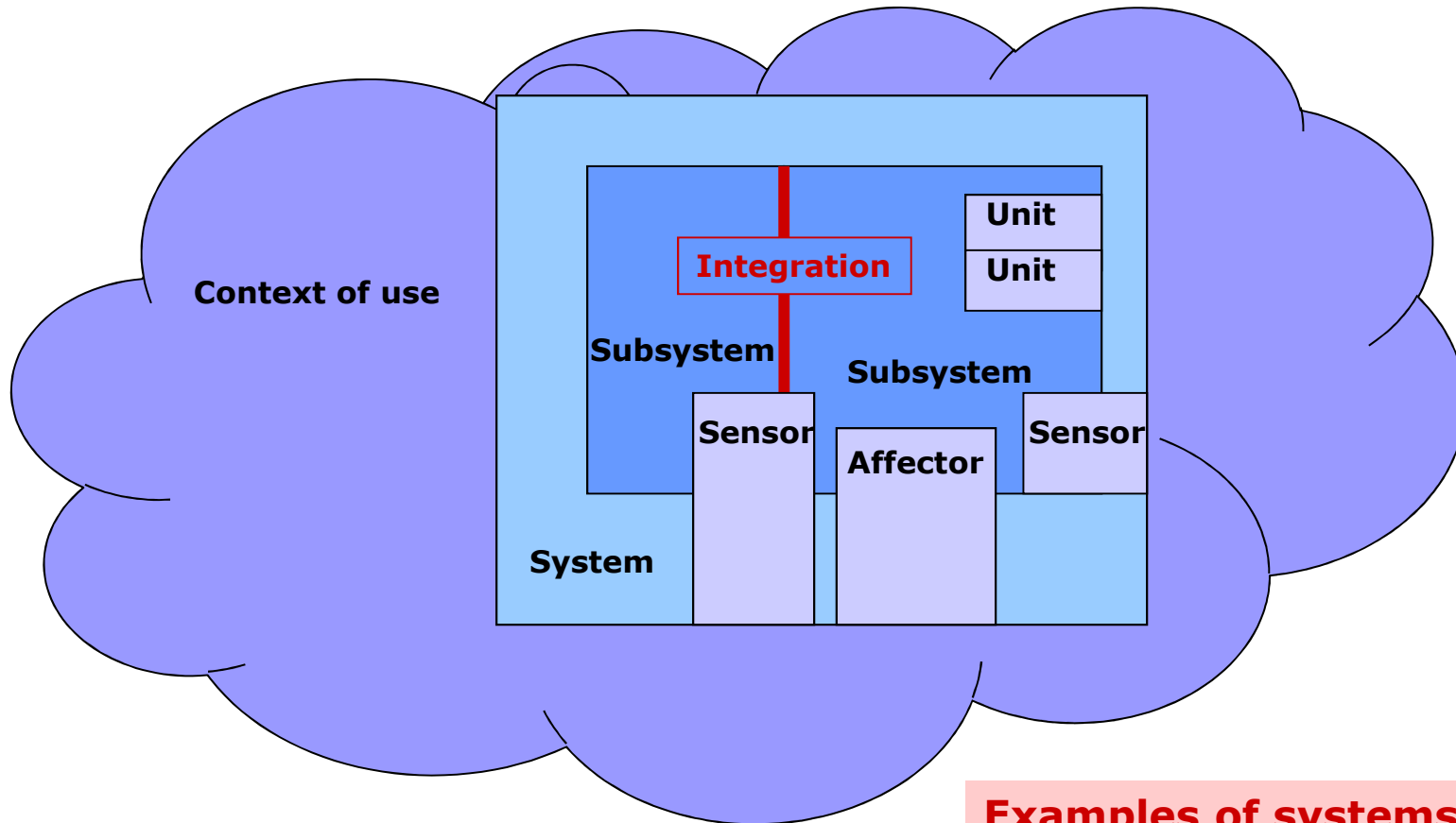
## 5. What are effective testing practices?

- 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*

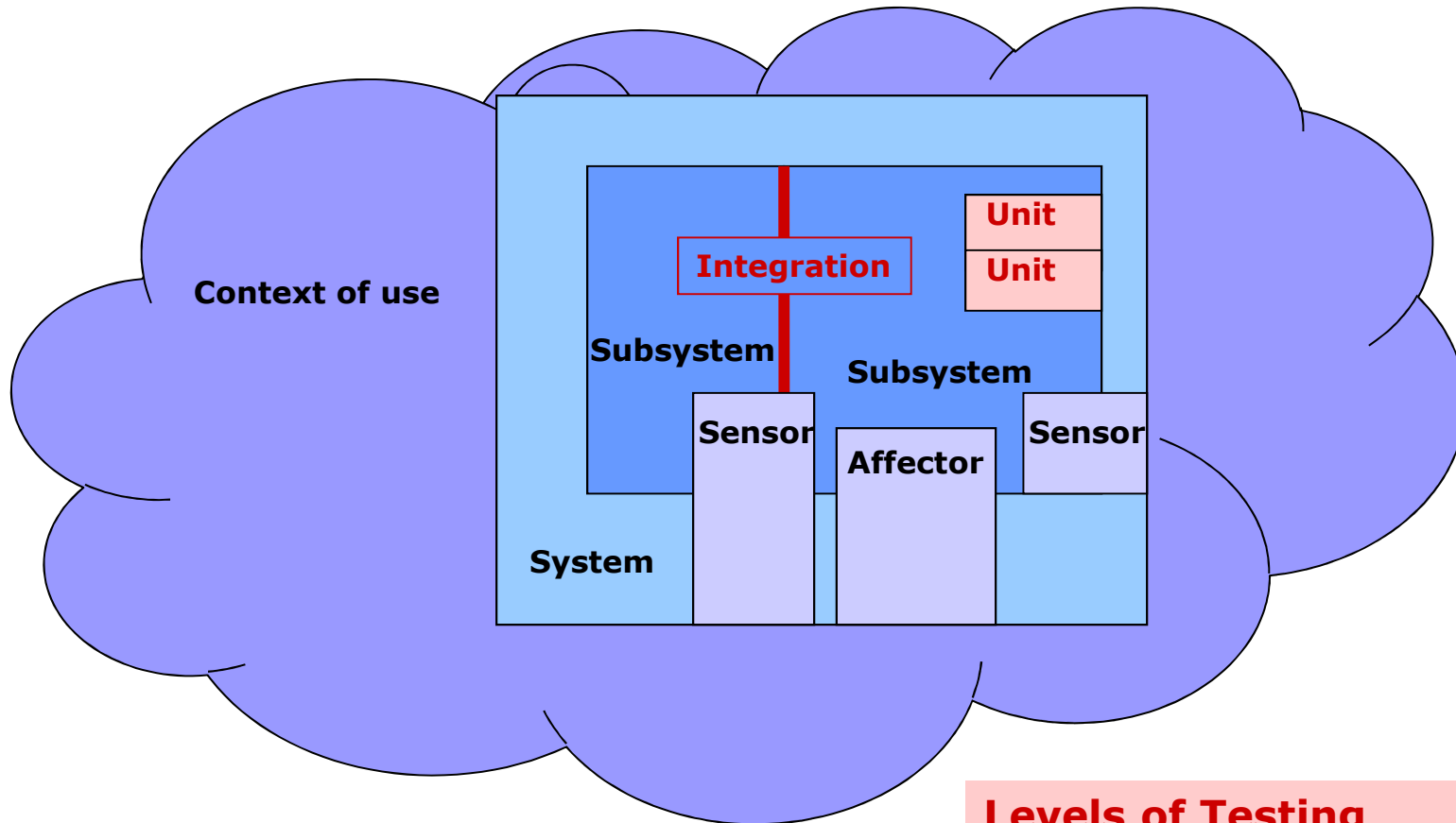
## 5. What do we test – the Focus of Concern



### 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

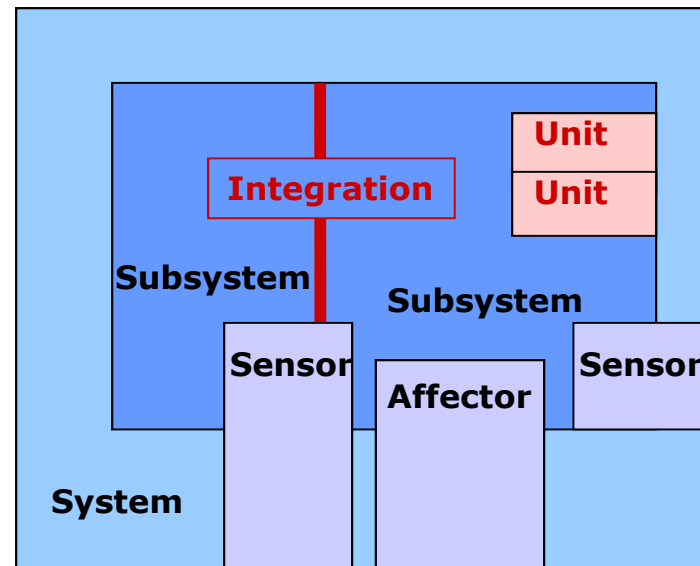


## Levels of Testing

- User testing, field testing



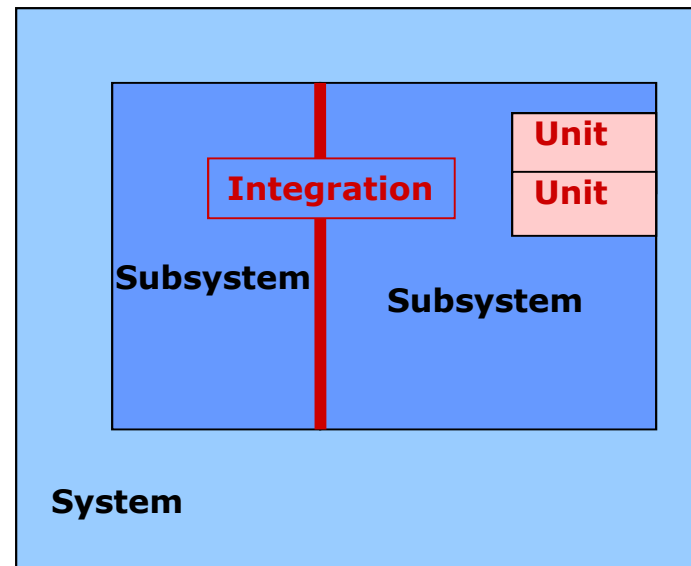
# The Focus of Concern



## Levels of Testing

- User testing, field testing
- System testing

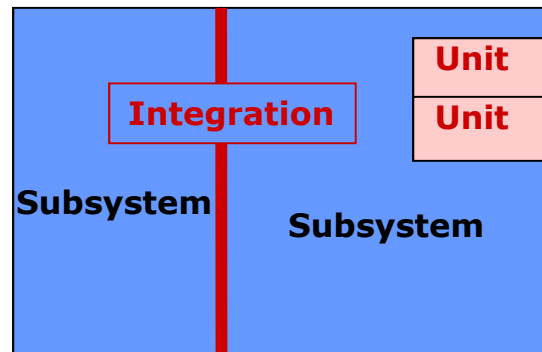
# The Focus of Concern



## 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]



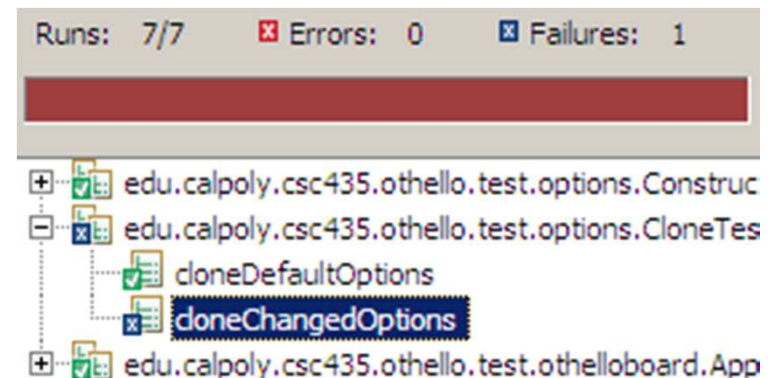
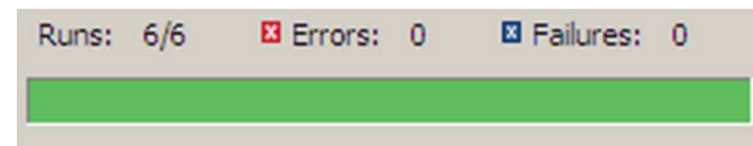
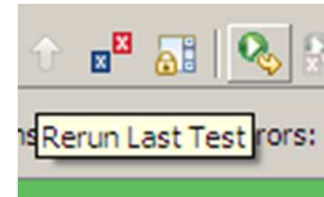
# JUnit: A Java Unit Testing Harness

- 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...

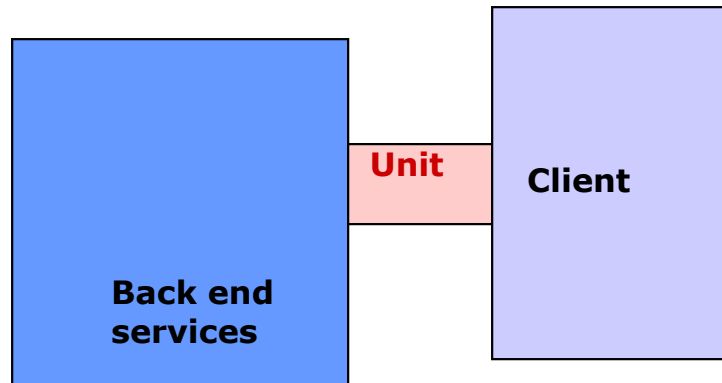
## 5. What are effective testing practices?

- Levels of structure: unit, integration, system...
- Design for testing: scaffolding
- Effective testing practices
- How does testing integrate into lifecycle and metrics?

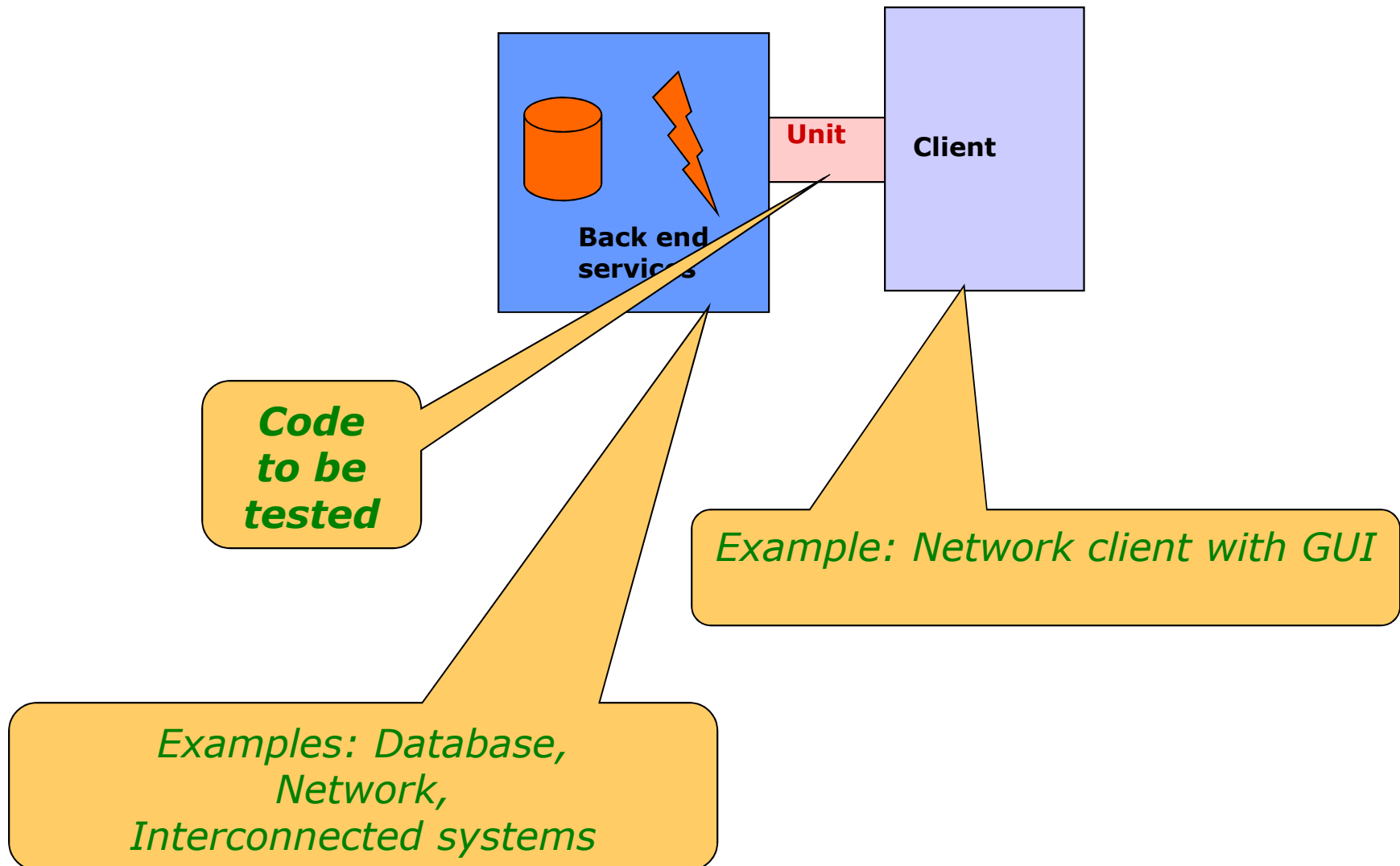
## 6. What are the limits of testing?

- What are complementary approaches?
  - *Inspections*
  - *Static and dynamic analysis*

# Unit Test and Scaffolding

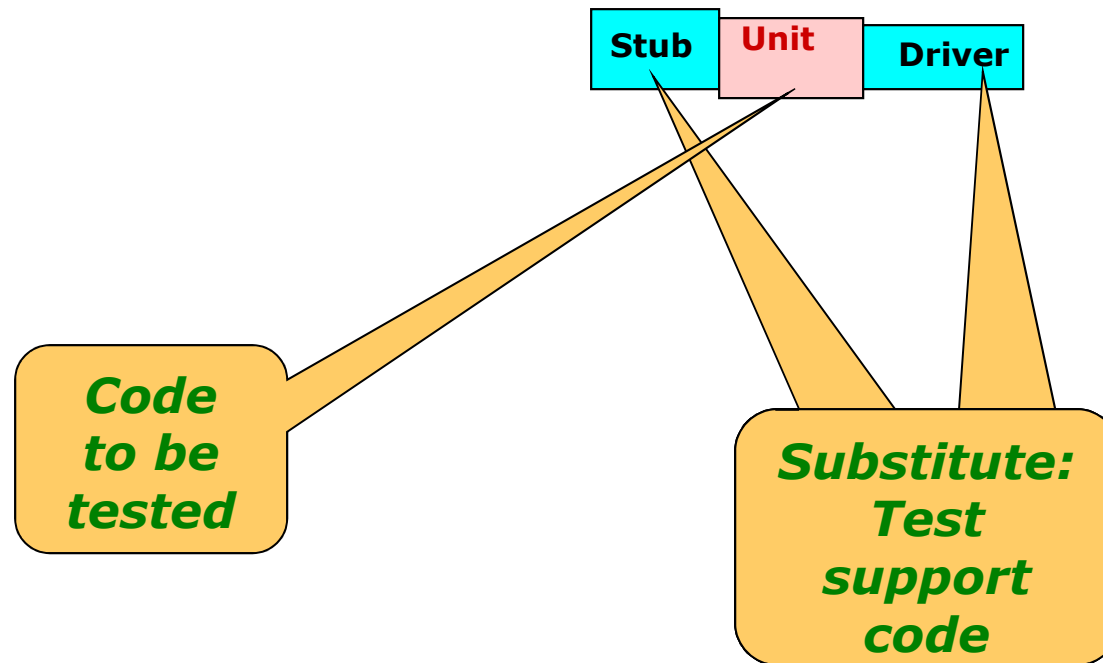


# Unit Test and Scaffolding

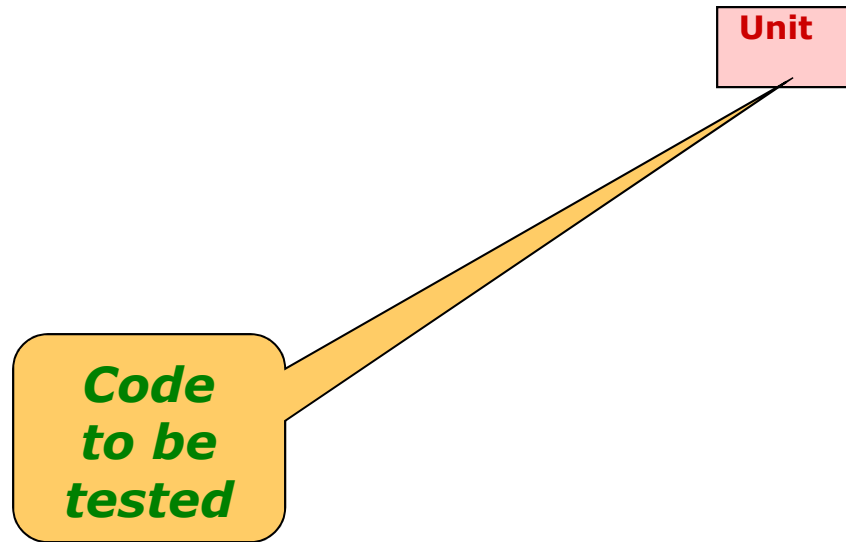




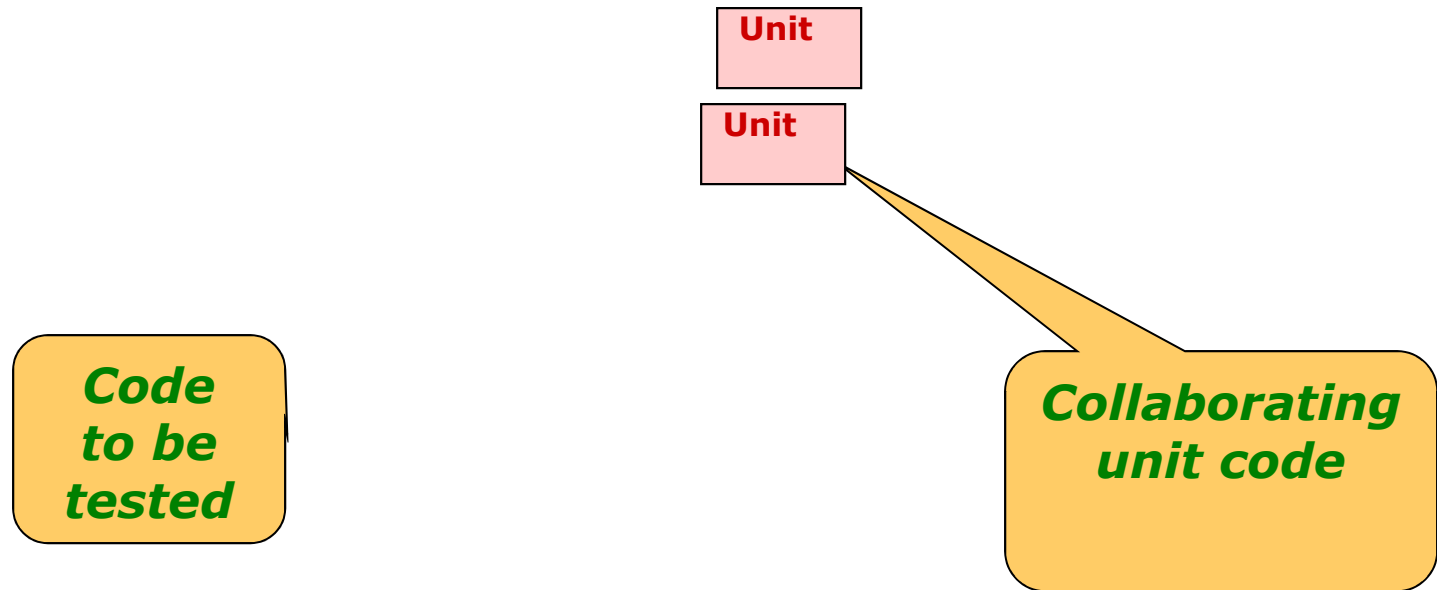
# Unit Test and Scaffolding



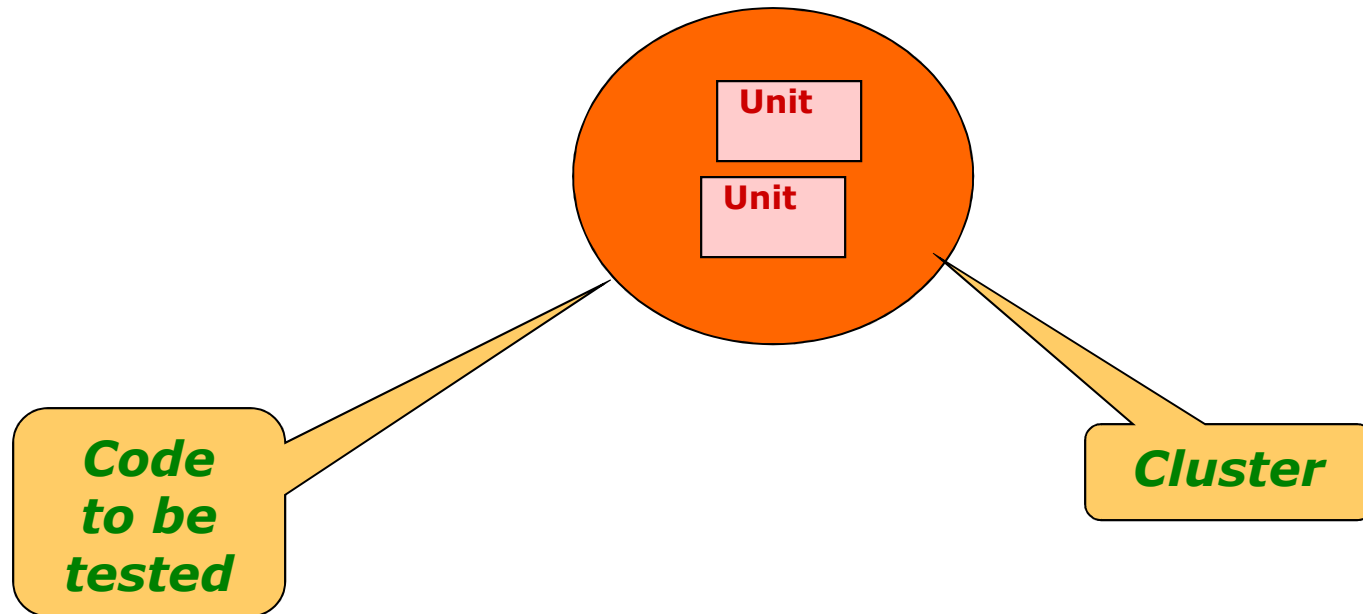
# Unit Test and Scaffolding



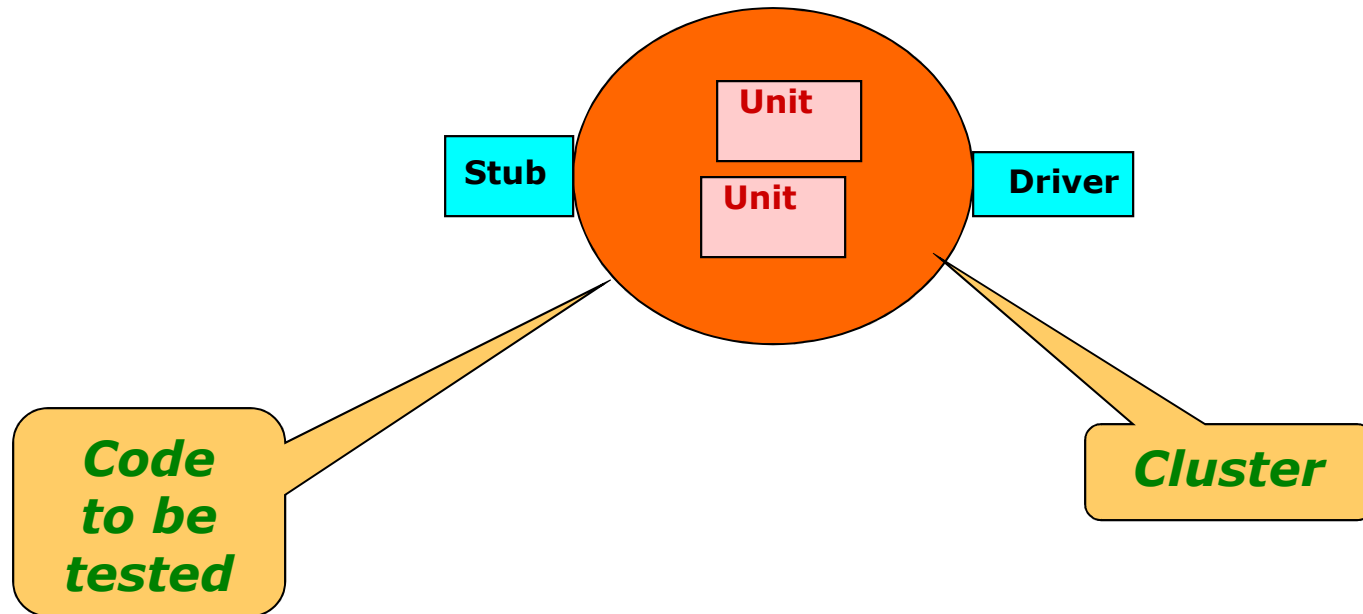
# Unit Test and Scaffolding



# Unit Test and Scaffolding



# Unit Test and Scaffolding



# Techniques for Unit Testing 1: Scaffolding

- Use “scaffold” to simulate external code

- External code – scaffold points

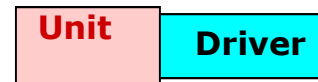
1. Client code
2. Underlying service code

## 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

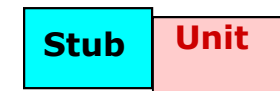


# Techniques for Unit Testing 1: Scaffolding

- 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



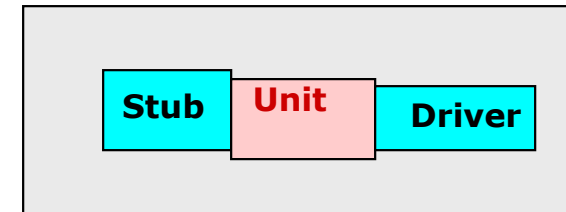
Testers write  
stub code



# 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

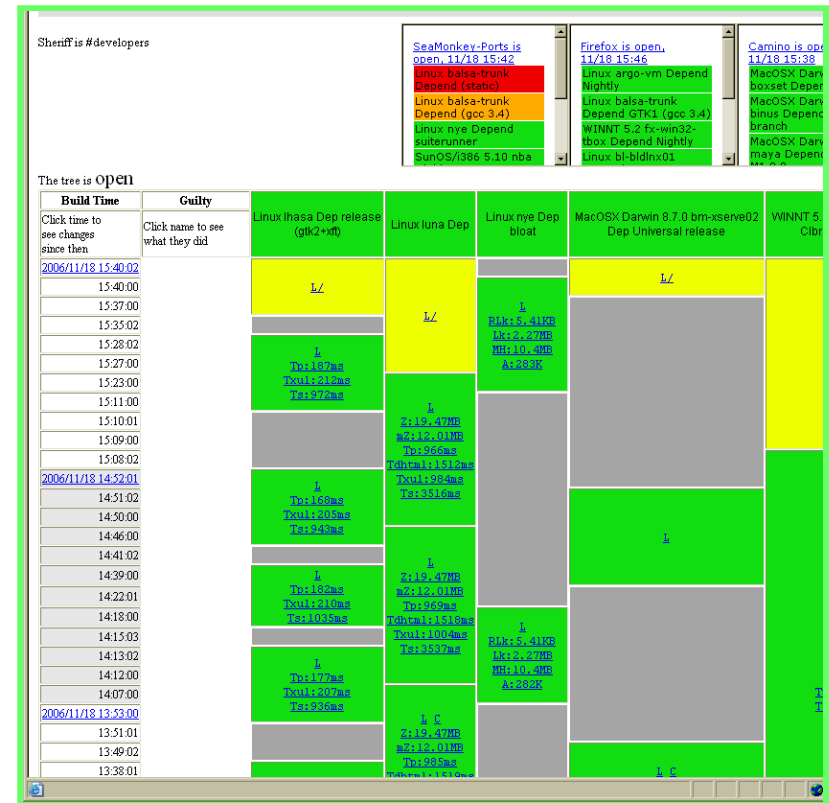
- Set up subsystem for test
- Test specific subsystem- or system-level features
  - no “random input” sequence
- Verify expected output



## 5c. Frequent (Nightly) Builds

### 2. Build a release of a large project every night

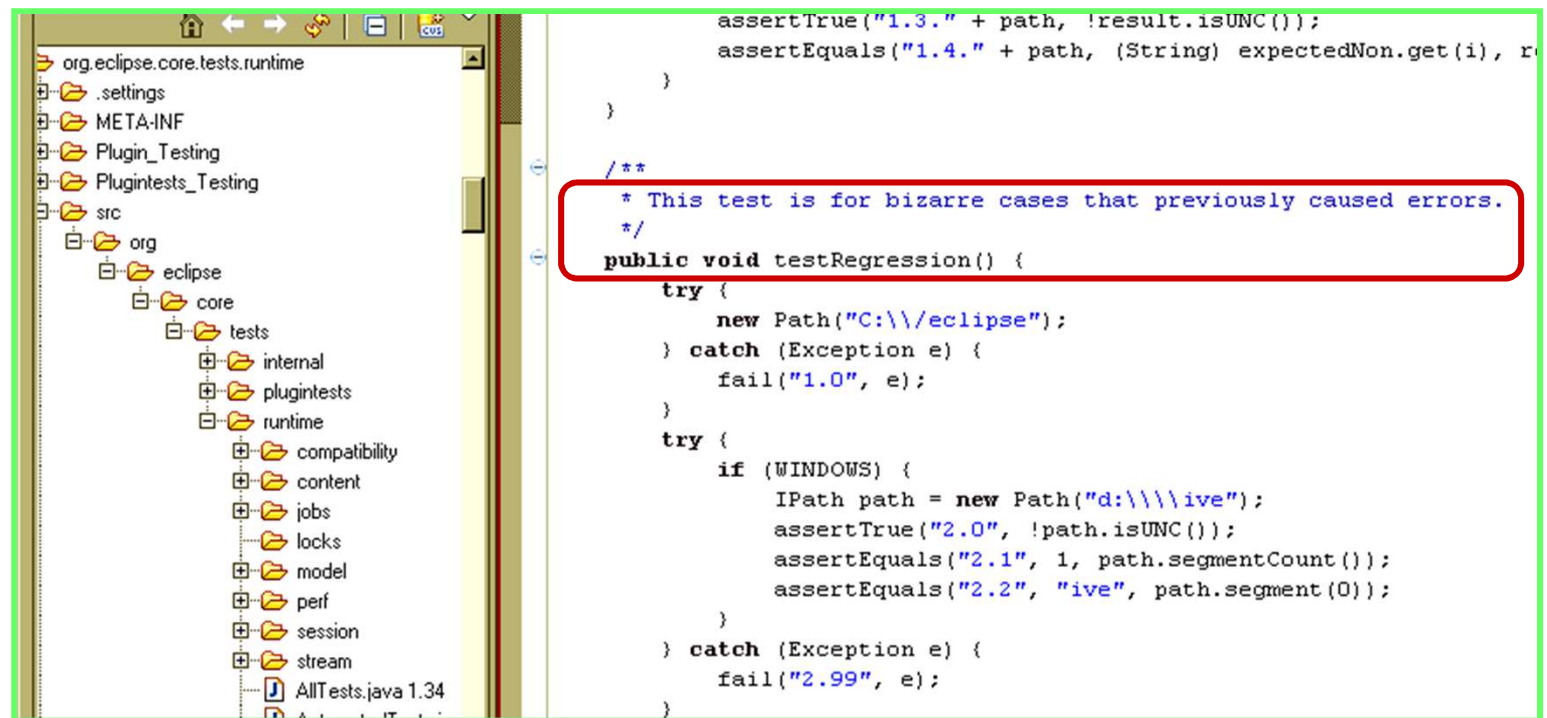
- Catches integration problems where a change “breaks the build”
  - Breaking the build is a BIG deal—may result in midnight calls to the responsible engineer
- Use test automation
  - Upfront cost, amortized benefit
  - Not all tests are easily automated – manually code the others
- Run simplified “smoke 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

The screenshot displays the Eclipse Bugzilla interface for bug 141261. The header shows the Eclipse logo and 'Eclipse bugs Bugzilla 3.20.3'. The bug title is 'crash - Shell create, RepositionWindow() - Unexpected Eclipse crash RC3 (JavaNativeCrash)'. The last modified date is 2006-11-14 17:46:58. The bug list shows 31 of 200 bugs. The form includes fields for Bug#, Product, Component, Status, Resolution, Assigned To, Hardware, OS, Version, Priority, Severity, Target Milestone, Reporter, Add CC, CC, QA Contact, URL, Summary, Status, Whiteboard, and Keywords. The bug is assigned to Silenio Quarti. The summary is 'crash - Shell create, RepositionWindow() - Unexpected Eclipse c'. The status is 'NEW'. The whiteboard is empty. The keywords are empty. The attachment table shows one attachment: 'Create a New Attachment (proposed patch, testcase, etc.)'. The bug depends on no other bugs and blocks no other bugs. The votes are 0. The additional comments section is empty.

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

Bug List: (31 of 200) [First](#) [Last](#) [Prev](#) [Next](#) [Show last search results](#) [Search page](#) [Enter new bug](#)

**[Eclipse] Bug#:** 141261 **Hardware:** Macintosh **Reporter:** Igor Goldenberg <igorg@gigaspace.com>

**Product:** Platform **OS:** Mac OS **Add CC:**

**Component:** SWT **Version:** 3.2 **Priority:** P3 **Severity:** major

**Status:** NEW **Assigned To:** Silenio Quarti <Silenio\_Quarti@ca.ibm.com> **Target Milestone:**

**QA Contact:** **URL:** **Summary:** crash - Shell create, RepositionWindow() - Unexpected Eclipse c

**Status:** **Whiteboard:** **Keywords:**

Attachment	Type	Created	Size	Actions
<a href="#">Create a New Attachment</a> (proposed patch, testcase, etc.)				<a href="#">View All</a>

Bug 141261 depends on: **Bug 141261 blocks:** [Show dependency tree](#)

**Votes:** 0 [Show votes for this bug](#) [Vote for this bug](#)

**Additional Comments:**

# 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

☐ [Reassign](#) bug to

☐ Reassign bug to default assignee and QA contact of selected component

[View Bug Activity](#) | [Format For Printing](#) | [Clone This Bug](#)

---

**Description:** [\[reply\]](#) Opened: 2005-07-25 07:03

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

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

This is actually two bugs imho;

1. The details is very very poor so I really don't know what happened. A stacktrace would be great for debugging.
2. The undo obviously does not work correctly.

here is a screenshot of the crash:  
[http://mnemo.minimum.se/eclipse\\_crashes/eclipse\\_undo\\_crash.png](http://mnemo.minimum.se/eclipse_crashes/eclipse_undo_crash.png)

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



## Practices: Root cause analysis

### 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

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

- Make decisions regarding when to unit test

- There are differing views
  - **CleanRoom**: Defer testing. Use separate test team
  - ✓ • **Agile**: As early as possible, even before code, integrate into team

# Test Strategy Statement

- Examples:

- We will release the product to friendly users after a brief internal review to find any truly glaring problems. The friendly users will put the product into service and tell us about any changes they'd like us to make.
- We will define use cases in the form of sequences of user interactions with the product that represent ... the ways we expect normal people to use the product. We will augment that with stress testing and abnormal use testing (invalid data and error conditions). Our top priority is finding fundamental deviations from specified behavior, but we will also use exploratory testing to identify ways in which this program might violate user expectations.
- We will perform parallel exploratory testing and automated regression test development and execution. The exploratory testing will focus on validating basic functions (capability testing) to provide an early warning system for major functional failures. We will also pursue high-volume random testing where possible in the code.

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

# Why Produce a Test Plan?

## 4. Ensure the test plan addresses the needs of stakeholders

- Customer: may be a required product
  - Customer requirements for operations and support
  - Examples
    - Government systems integration
    - Safety-critical certification: avionics, health devices, etc.
- A separate test organization may implement part of the plan
  - “IV&V” – Independent verification and validation
- May benefit development team
  - Set priorities
    - Use planning process to identify areas of hazard, risk, cost
- Additional benefits – the plan is a team product
  - Test quality
    - Improve coverage via list of features and quality attributes
    - Analysis of program (e.g. boundary values)
    - Avoid repetition and check completeness
  - Communication
    - Get feedback on strategy
    - Agree on cost, quality with management
  - Organization
    - Division of labor
    - Measurement of progress

# Defect Tracking

## 5. Track defects and issues

- **Issue: Bug, feature request, or query**
  - May not know which of these until analysis is done, so track in the same database (Issuezilla)
- **Provides a basis for measurement**
  - Defects reported: which lifecycle phase
  - Defects repaired: time lag, difficulty
  - Defect categorization
  - Root cause analysis (more difficult!)
- **Provides a basis for division of effort**
  - Track diagnosis and repair
  - Assign roles, track team involvement
- **Facilitates communication**
  - Organized record for each issue
  - Ensures problems are not forgotten
- **Provides some accountability**
  - Can identify and fix problems in process
    - Not enough detail in test reports
    - Not rapid enough response to bug reports
  - Should not be used for HR evaluation

----- Comment #4 From [Clare Carty](#) 2006-10-11 15:28 [reply] -----

(In reply to [comment #3](#))  
> I'm sorry but we really don't have enough details to be able  
> problem. Could you try with another VM?  
>  
Problem didn't happen with another JRE - just the sun JRE.

----- Comment #5 From [Oleg Besedin](#) 2006-10-11 15:38 [reply] -----

This looks like a duplicate of the [bug 92250](#). Could you try if with -XX:MaxPermSize=256m ?

----- Comment #6 From [Pascal Rapiçault](#) 2006-10-12 12:57 [reply] -----

After further investigation, setting the permgenspace to 1024 r problem.  
\*\*\* This bug has been marked as a duplicate of [92250](#) \*\*\*

----- Comment #7 From [Clare Carty](#) 2006-10-12 15:18 [reply] -----

This problem is still occurring on the dependent product with H to 1024H. Please investigate.

----- Comment #8 From [John Arthorne](#) 2006-10-12 17:24 [reply] -----

What version of the Sun JRE are you using? I suggest trying w later, as there are known memory leak problems with 1.5.0\_06 or

Bug List: (48 of 200) [First](#) [Last](#)

[Eclipse] Bug#: [160502](#) Hardware:  OS:  Reporter: [Clare Carty](#)  
Product:  Version:  Add CC:   
Component:  Priority:  CC: [ccarty@ca.ibm.com](#)  
Status: REOPENED Severity:  [john\\_arthorne@ca.ibm.com](#)  
Resolution:  Target:  ☐ Remove selected CCs  
Assigned To:  Milestone:

QA Contact:   
URL:   
Summary:   
Status:   
Whiteboard:   
Keywords:

Attachment	Type	Created	Size	Actions
<a href="#">screenshot of crash</a>	image/jpeg	2006-10-11 12:14	131.55 KB	<a href="#">Edit</a>
<a href="#">Create a New Attachment</a> (proposed patch, testcase, etc.)				<a href="#">View All</a>

Bug 160502 depends on:  [Show dependency tree](#)  
Bug 160502 blocks:

Votes: 0 [Show votes for this bug](#) [Vote for this bug](#)



# 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*

## 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?