Software Testing & Unit Tests

• Resources
  • Ian Sommerville
    Software Engineering 8th Edition
    Addison Wesley 2007
  • Robert v. Binder
    Testing Object-Oriented Systems - Models, Patterns, and Tools
    Addison Wesley 2000
Software Testing
Validation

“Are we building the *right* product?”

Verification

“Are we *building* the *product* right?”

Ian Sommerville

*Software Engineering 8th Edition; Addison Wesley 2007*
Two complementary approaches for verification and validation (V&V) can be distinguished.

• **Software Inspections or Peer Reviews**
  (Static Technique)
  “Software inspections” can be done at all stages of the process.

• **Software Testing**
  (Dynamic Technique)
Software inspections check the correspondence between a program and its specification.

- Some techniques

- **Program inspections**
  The goal is to find program defects, standards violations, poor code rather than to consider broader design issues; it is usually carried out by a team and the members systematically analyze the code. An inspection is usually driven by checklists. (Studies have shown that an inspection of roughly 100LoC takes about one person-day of effort.)

- ...

Software Inspections - Static Technique | 5
Software inspections check the correspondence between a program and its specification.

- Some techniques
- ...

- **Automated source code analysis**
  Includes - among others - control flow analysis, data use / flow analysis, information flow analysis and path analysis.
  Static analyses draw attention to anomalies.

- ...

---

Software Inspections - Static Technique | 6
Software inspections check the correspondence between a program and its specification.

• Some techniques

• ...

• **Formal verification**
  Formal verification can guarantee the absence of specific bugs. E.g., to guarantee that a program does not contain deadlocks, race conditions or buffer overflows.
Software inspections check the correspondence between a program and its specification.

Software inspections do not demonstrate that the software is useful.
Software testing refers to running an implementation of the software with test data to discover program defects.

- **Validation testing**
  Intended to show that the software is what the customer wants (Basically, there should be a test case for every requirement.)

- **Defect testing**
  Intended to reveal defects
- (Defect) Testing is...
  - fault directed when the intent is to reveal faults
  - conformance directed when the intent is to demonstrate conformance to required capabilities

No Strict Separation
Test plans set out the testing schedule and procedures; they establish standards for the testing process. They evolve during the development process.

- V&V is expensive; sometimes half of the development budget is spent on V&V
The scope of a test is the collection of software components to be verified.

- **Unit tests**
  (dt. Modultest)
  Comprises a relatively small executable; e.g., a single object

- **Integration test**
  Complete (sub)system. Interfaces among units are exercised to show that the units are collectively operable

- **System test**
  A complete integrated application. Categorized by the kind of conformance they seek to establish: functional, performance, stress or load
Testing can only show the presence of errors, not their absence.

E. Dijkstra
The design of tests is a multi-step process.

1. Identify, model and analyze the responsibilities of the system under test (SUT)  
   (E.g., use pre- and postconditions identified in use cases as input.)

2. Design test cases based on this external perspective

3. Add test cases based on code analysis, suspicions, and heuristics

4. Develop expected results for each test case or choose an approach to evaluate the pass / no pass status of each test case
A test automation system (TAS) will be used to execute the tests.

A test automation system will...

• start the implementation under test (IUT)
• set up its environment
• bring it to the required pretest state
• apply the test inputs
• evaluate the resulting output and state

Complex systems often require a significant customization of existing test automation systems.
The goal of the test execution is to establish that the implementation under test (IUT) is minimally operational by exercising the interfaces between its parts.

To establish the goal...

1. execute the test suite; the result of each test is evaluated as pass or no pass
2. use a coverage tool to instrument the implementation under test; rerun the test suite and evaluate the reported coverage
3. if necessary, develop additional tests to exercise uncovered code
4. stop testing when the test goal is met; all tests pass
   ("Exhaustive" testing is generally not possible!)
**Test Point**
(dt. Testdatum (Prüfpunkt))

- A test point is a specific value for...
  - test case input
  - a state variable
- The test point is selected from a domain; the domain is the set of values that input or state variables may take

- Heuristics for test point selection:
  - Equivalence Classes
  - Boundary Value Analysis
  - Special Values Testing
Test Case
(dt. Testfall)

- Test cases specify:
  - pretest state of the implementation under test (IUT)
  - test inputs / conditions
  - expected results
Test Suite

• A test suite is a collection of test cases
Test Run
(dt. Testlauf)

• A test run is the execution (with results) of a test suite
• The IUT produces actual results when a test case is applied to it; a test whose actual results are the same as the expected results is said to pass
Test Driver

&

Test Harness/Automated Test Framework

- Test driver is a class or utility program that applies test cases to an IUT
- Test harness is a system of test drivers and other tools to support test execution
Failures, Errors & Bugs

Failure = dt. Defekt(, Fehlschlag)
Fault = dt. Mangel
Error = dt. Fehler

- A **failure** is the (manifested) inability of a system or component to perform a required function within specified limits
- A **software fault** is missing or incorrect code
- An **error** is a human action that produces a software fault
- **Bug**: error or fault.
Test Plan

- A document prepared for human use that explains a testing approach:
  - the work plan,
  - general procedures,
  - explanation of the test design,
  - ...
Testing must be based on a **fault model**.

Because the number of tests is infinite, we have to make (for practical purposes) an assumption about **where faults are likely to be found**!
Testing must be based on a fault model.

Two general fault models and corresponding testing strategies exist:

- Conformance-directed testing
- Fault-directed testing

Testing has to be efficient.
Developing a Test Plan

• Devise a test plan for a program that:
  • reads three integer values,
  • which are interpreted as the length of the sides of a triangle
• The program states whether the triangle is
  • scalene (dt. schief),
  • isosceles (dt. gleichschenklig), or
  • equilateral (dt. gleichseitig)

• A valid triangle must meet two conditions:
  • No side may have a length of zero
  • Each side must be shorter than the sum of all sides divided by 2
An Implementation of a Triangle

```java
class Polygon extends Figure {
    abstract void draw(...);
    abstract float area();
}

class Triangle extends Polygon {
    public Triangle(...);
    public void setA(LineSegment a);
    public void setB(LineSegment b);
    public void setC(LineSegment c);
    public boolean isIsosceles();
    public boolean isScalene();
    public boolean isEquilateral();
}
```
## Test Descriptions

<table>
<thead>
<tr>
<th>Description</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>Expected Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valid scalene triangle</td>
<td>5</td>
<td>3</td>
<td>4</td>
<td>Scalene</td>
</tr>
<tr>
<td>Valid isosceles triangle</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>Isosceles</td>
</tr>
<tr>
<td>Valid equilateral triangle</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>Equilateral</td>
</tr>
<tr>
<td>First perm. of two equal sides</td>
<td>50</td>
<td>50</td>
<td>25</td>
<td>Isosceles</td>
</tr>
<tr>
<td><em>(Permutations of previous test case)</em></td>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>One side zero</td>
<td>1000</td>
<td>1000</td>
<td>0</td>
<td>Invalid</td>
</tr>
<tr>
<td>First perm. of two equal sides</td>
<td>10</td>
<td>5</td>
<td>5</td>
<td>Invalid</td>
</tr>
<tr>
<td>Sec. perm. of two equal sides</td>
<td>5</td>
<td>10</td>
<td>5</td>
<td>Invalid</td>
</tr>
<tr>
<td>Third perm. of two equal sides</td>
<td>5</td>
<td>5</td>
<td>10</td>
<td>Invalid</td>
</tr>
<tr>
<td>Three sides greater than zero, sum of two smallest less than the largest</td>
<td>8</td>
<td>5</td>
<td>2</td>
<td>Invalid</td>
</tr>
</tbody>
</table>
## Test Descriptions

<table>
<thead>
<tr>
<th>Description</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>Expected Output</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>(Permutations of previous test case)</em></td>
<td>...</td>
<td>...</td>
<td>...</td>
<td><em>Invalid</em></td>
</tr>
<tr>
<td>All sides zero</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td><em>Invalid</em></td>
</tr>
<tr>
<td>One side equals the sum of the other</td>
<td>12</td>
<td>5</td>
<td>7</td>
<td><em>Invalid</em></td>
</tr>
<tr>
<td><em>(Permutations of previous test case)</em></td>
<td>...</td>
<td>...</td>
<td>...</td>
<td><em>Invalid</em></td>
</tr>
<tr>
<td>Three sides at maximum possible value</td>
<td>MAX</td>
<td>MAX</td>
<td>MAX</td>
<td><em>Equilateral</em></td>
</tr>
<tr>
<td>Two sides at maximum possible value</td>
<td>MAX</td>
<td>MAX</td>
<td>1</td>
<td><em>Isosceles</em></td>
</tr>
<tr>
<td>One side at maximum value</td>
<td>1</td>
<td>1</td>
<td>MAX</td>
<td><em>Invalid</em></td>
</tr>
</tbody>
</table>

+ Further OO related tests w.r.t. the type hierarchy etc.  
(e.g. are the line segments connected.)
Developing a Test Plan

Let’s assume that we are going to write a tool for verifying Java code. In particular, we would like to assert that specific int based calculations always satisfies the stated assertion.

```java
public int doCalc(int i, int j) {
    if (i < 0 || i > 10 || j < 0 || j > 100)
        throw new IllegalArgumentException();

    return i * j; //ASSERT(i * j in [0,1000])
}
```
Developing a Test Plan

To represent Java int values, we are using the following classes and map the calculations to the respective methods.

```java
/** Representation of a primitive Java int value. */
abstract class IntValue {

    /**
     * Calculates the result of multiplying a and b. The result is as precise as possible given
     * the available information. If the result is either a or b, the respective object is
     * returned.
     */
    public abstract IntValue mul(IntValue other);
}

/** Represents a specific but unknown Java int value. */
class AnInt extends IntValue {

    public IntValue mul(IntValue other) {...}
}

/** Represents a value that is in the range [lb,ub]; however, the specific value is unknown. */
class Range extends IntValue {

    public final int lb;
    public final int ub;

    public Range(int lb, int ub) {
        this.lb = lb;
        this.ub = ub;
    }

    public IntValue mul(IntValue other) {...}
}
```

How does the test plan look like?
The Control-flow Graph of a Method

A representation of all paths through a program.

```
static void doThat(int v, boolean b) {
    if (v > 100 && b) {
        print("if");
    }
    else {
        print("else");
    }
    return;
}
```
Common Method Scope Code Coverage Models

• **Statement Coverage** is achieved when all statements in a method have been executed at least once

• **Branch Coverage** is achieved when every path from a node is executed at least once by a test suite; compound predicates are treated as a single statement

• **Simple Condition Coverage** requires that each simple condition be evaluated as true and false at least once (Hence, it does not require testing all possible branches.)

• **Condition Coverage** = **Simple Condition Coverage + Branch Coverage**

• **Multiple-condition Coverage** requires that all true-false combinations of simple conditions be exercised at least once

• **Modified condition/decision coverage** (Recommended for, e.g., SIL 4 Software)

branch = dt. Verzweigung; condition = dt. Bedingung;
branch coverage = dt. Zweigüberdeckung
simple condition coverage = dt. einfache Bedingungsüberdeckung
Conditions - Exemplified

```java
static void doThat(int v, boolean b) {
    simple/atomic condition(s)
    if (v > 100 && b) {
        print("if");
    } else {
        print("else");
    }
}
```

Here, "v > 100" is the first condition and "b" is the second condition.

In Java, simple/atomic conditions are separated by "&&" / "&" or "||" / "|" operators.
Compound Predicates - Exemplified

Here, “v > 100 && b” is called a predicate resp. a compound predicate. This compound predicate consists of two “simple” conditions.

```java
static void doThat(int v, boolean b) {
    (compound) predicate (expression)

    if (v > 100 && b) {
        print("if");
    }
    else {
        print("else");
    }
}
```
Branch Coverage Exemplified

When we have shortcut evaluation, simple condition coverage implies branch coverage!
Simple Condition Coverage Exemplified

Recall: The condition is an expression that evaluates to true or false. I.e., an expression such as !b (not b) is the condition.

```
static void doThat(
    boolean a,
    boolean b,
    boolean c) {
    if ((a & c) | (c & b) | (b & a)) {
        print("if");
    } else {
        print("else");
    }
}
```

100% Simple Condition Coverage
- a = true, b = false, c = false
- a = false, b = true, c = false
- a = false, b = false, c = true

Using “plain” logical boolean operators!
(Simple) Condition Coverage Exemplified

100% (Simple) Condition Coverage
a = true, c = true (b is not relevant)
a = false, c = true, b = true
a = false, c = false, b = false

static void doThat(
    boolean a,
    boolean b,
    boolean c) {
    if ((a && c) || (c && b) || (b && a)) {
        print("if");
    } else {
        print("else");
    }
}

Recall, if we have shortcut evaluation, simple condition coverage implies branch coverage!

Using conditional-and/or operators!
Basic Block Coverage

• A basic block is a sequence of consecutive instructions in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end.

• Basic block coverage is achieved if all basic blocks of a method are executed

(⚡ Sometimes “statement coverage” is used as a synonym for “basic block coverage” - however, we do not use these terms synonymously.)
(Basic blocks are sometimes called segments.)
Basic Block Coverage Exemplified

100% Basic Block Coverage

v = 90, b = "not relevant"

v = 101, b = true

```java
static void doThat(int v, boolean b) {
    if (v > 100 && b) {
        print("if");
    } else {
        print("else");
    }
}
```

At the bytecode level (or lower), no explicit support for conditional boolean operators exists and are therefore compiled using respective "if"s.

This graph is the control-flow graph that compilers typically generate when compiling the source code shown on the left hand side.
static void doThis(boolean a, boolean b) {
    if (a) {
        print("A");
    }
    if (b) {
        print("B");
    }
}

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>Minimal Number of Tests to Achieve Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Statement</strong></td>
<td>TRUE</td>
<td>TRUE</td>
<td></td>
</tr>
<tr>
<td><strong>Basic Block</strong></td>
<td>TRUE</td>
<td>TRUE</td>
<td></td>
</tr>
<tr>
<td><strong>(Simple) Condition Coverage</strong></td>
<td>TRUE</td>
<td>TRUE</td>
<td></td>
</tr>
<tr>
<td><strong>Branch Coverage</strong></td>
<td>FALSE</td>
<td>FALSE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FALSE</td>
<td>FALSE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TRUE</td>
<td>TRUE</td>
<td></td>
</tr>
</tbody>
</table>

Here, condition coverage can also be achieved using other test cases. (E.g. a=false; b=true and a=true; b=false.)

No case covers all possible paths!
static void doThis(boolean a, boolean b) {
    if (a && b) {
        print("A && B");
    }
}

### Table: Minimal Number of Tests to Achieve Coverage

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statement</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
<tr>
<td>Basic Block</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
<tr>
<td>(Simple) Condition Coverage</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
<tr>
<td></td>
<td>TRUE</td>
<td>FALSE</td>
</tr>
<tr>
<td></td>
<td>FALSE</td>
<td>/</td>
</tr>
<tr>
<td>Branch Coverage</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
<tr>
<td>(w.r.t. the given source code)</td>
<td>FALSE</td>
<td>/</td>
</tr>
<tr>
<td>Multiple Condition Coverage</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
<tr>
<td></td>
<td>TRUE</td>
<td>FALSE</td>
</tr>
<tr>
<td></td>
<td>FALSE</td>
<td>/</td>
</tr>
</tbody>
</table>

**Question:**
*Würde im Falle von Condition Coverage nicht auch "true, true" und "false, false" ausreichen?*

**Answer:**
*Da im Ausdruck "a && b", "b" nur evaluiert wird wenn a wahr ist (Short-circuit Evaluation von "&&" - siehe Graph) - ist "false / false" keine hilfreiche Belegung der Parameter.*

---

**Question / Antwort:**
*Wäre der Code:
if (a) {
    if (b)
        print("A && B")
    else
        print("Hello!")
} else {
    print("Hello!")
} return;*  
*dann wäre für "Statement Coverage" folgende Testfälle notwendig: a=true; b=false und a=true; b=true. (Ebenso für Basic Block Coverage)*
static void doThis(boolean a, boolean b) {
    if (a || b) {
        print("A or B");
    }
}

We have achieved 100% statement coverage, though we have never evaluated the condition b.

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statement</td>
<td>TRUE</td>
<td>/</td>
</tr>
<tr>
<td>Basic Block</td>
<td>FALSE</td>
<td>TRUE</td>
</tr>
<tr>
<td>(Simple)</td>
<td>FALSE</td>
<td>TRUE</td>
</tr>
<tr>
<td>Condition Coverage</td>
<td>FALSE</td>
<td>FALSE</td>
</tr>
<tr>
<td>Branch Coverage</td>
<td>TRUE</td>
<td>/</td>
</tr>
<tr>
<td>(w.r.t. the source code)</td>
<td>TRUE</td>
<td>/</td>
</tr>
<tr>
<td></td>
<td>FALSE</td>
<td>FALSE</td>
</tr>
</tbody>
</table>

static void doThis(boolean a, boolean b)
static long process(String[] args) throws IllegalArgumentException {

Stack values = new Stack();
for (int i = 0; i < args.length; i++) {
    String arg = args[i];
    try {
        long value = Long.parseLong(arg);
        values.push(value);
    } catch (NumberFormatException nfe) {
        // there is no method to test if a string is a number ...

        if (values.size() > 1) {
            long r = values.pop();
            long l = values.pop();
            if (arg.equals("+")) {
                values.push(l + r);
                continue;
            }
            if (arg.equals("*")) {
                values.push(l * r);
                continue;
            }
        } else 
            throw new IllegalArgumentException("Too few operands or operator unknown.");
    }
}

if (values.size() == 1) return values.pop();
else throw new IllegalArgumentException("Too few (0) or too many (>1) operands.");
}
static long process(java.lang.String[] args)

0 stack = new demo.SimpleCalculator.Stack
    stack.<init>()

1 p14 = Φ(0 \leftarrow 0, p65 \leftarrow 11)
p17 = args.length
if(p17 > p14)

2 p20 = args[p14]

3 long p23 = java.lang.Long.parseLong(p20)
    stack.push(p23)

5 java.lang.NumberFormatException
    int p30 = stack.size()
    if(1 <= p30)

8 long p40 = stack.pop()
    long p43 = stack.pop()
    p47 = p20.equals("+")
    if(p47)

9 p50 = p20.equals("-")
    if(p50)

12 long p58 = p43 + p40
    stack.push(p58)

10 long p54 = p43 * p40
    stack.push(p54)

11 int p65 = p14 + 1

6 p36 = new IllegalArgumentException
    p36.<init>("...")
    throw p36

16 p72 = new IllegalArgumentException
    p72.<init>("...")
    throw p72

17 p76 = stack.pop()
    return p76

Impossible Path
(If we have no arguments, the size of the stack will be zero.)
static long process(java.lang.String[] args)

0. stack = new demo.SimpleCalculator.Stack
   stack.<init>()

1. p14 = Φ(0←0,p65←11)
   p17 = args.length
   if(p17 > p14)

2. p20 = args[p14]

3. long p23 = java.lang.Long.parseLong(p20)
   stack.push(p23)

4. Handling Exceptions

5. java.lang.NumberFormatException
   int p30 = stack.size()
   if(1 >= p30)

6. p36 = new IllegalArgumentException
   p36.<init>('...')
   throw p36

7. long p40 = stack.pop()
   long p43 = stack.pop()
   p47 = p20.equals('+'')
   if(p47)

8. long p58 = p43 + p40
   stack.push(p58)

9. long p54 = p43 * p40
   stack.push(p54)

10. p72 = new IllegalArgumentException
    p72.<init>('...')
    throw p72

11. p76 = stack.pop()
    return p76
Do not use a code coverage model as a test model.

Do not rely on code coverage models to devise test suites. Test from responsibility models and use coverage reports to analyze test suite adequacy.

Covering some aspect of a method [...] is never a guarantee of bug-free software.

Robert V. Bender

Testing Object-Oriented Systems
Addison Wesley 2000
Steve Cornett
http://www.bullseye.com/coverage.html

• Recommended Reading

Code Coverage Analysis
This paper gives a complete description of code coverage analysis (test coverage analysis), a software testing technique.

By Steve Cornett. Copyright © Bullseye Testing Technology 1996-2008. All rights reserved. Redistribution in whole or in part is prohibited without permission.

Contents
• Introduction
• Structural Testing and Functional Testing
• The Premise
• Basic Metrics
  • Statement Coverage
  • Decision Coverage
  • Condition Coverage
  • Multiple Condition Coverage
  • Condition/Decision Coverage
  • Modified Condition/Decision Coverage
  • Path Coverage
• Other Metrics
  • Function Coverage
  • Call Coverage
  • Linear Code Sequence and Jump (LCSSA) Coverage
  • Data Flow Coverage
  • Object Code Branch Coverage
  • Loop Coverage
  • Race Coverage
  • Relational Operator Coverage
  • Weak Mutation Coverage
  • Table Coverage
• Comparing Metrics
• Coverage Goal for Release
• Intermediate Coverage Goals
• Summary
• References
Limits of Testing
Limits of Testing
The number of input and output combinations for trivial programs is already (very) large.

Assume that we limit points to integers between 1 and 10; there are $10^4$ possible ways to draw (a single) line.

Since a triangle has three lines we have $10^4 \times 10^4 \times 10^4$ possible inputs of three lines (including invalid combinations).

**We can never test all inputs, states, or outputs.**
Limits of Testing

Branching and (dynamic binding) result in a very large number of unique execution sequences. Simple iteration increases the number of possible sequences to astronomical proportions.

```
for (int i = 0; i < n; ++i)
{
    if (a.get(i) == b.get(i))
        x[i] = x[i] + 100;
    else
        x[i] = x[i] / 2;
}
```
**Limits of Testing**

Branching and dynamic binding result in a very large number of unique execution sequences.

- If we count entry-exit paths without regarding iteration there are only three paths:
  - loop header, exit
  - loop header, cond., +100
  - loop header, cond., /2
Limits of Testing

Branching and dynamic binding result in a very large number of unique execution sequences. Simple iteration increases the number of possible sequences to astronomical proportions.

<table>
<thead>
<tr>
<th>Number of iterations</th>
<th>Number of paths</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$2^1 + 1 = 3$</td>
</tr>
<tr>
<td>2</td>
<td>$2^2 + 1 = 5$</td>
</tr>
<tr>
<td>3</td>
<td>$2^3 + 1 = 9$</td>
</tr>
<tr>
<td>10</td>
<td>1.025</td>
</tr>
<tr>
<td>20</td>
<td>1.048.577</td>
</tr>
</tbody>
</table>

1. Path
Limits of Testing

Branching and dynamic binding result in a very large number of unique execution sequences. Simple iteration increases the number of possible sequences to astronomical proportions.

<table>
<thead>
<tr>
<th>Number of iterations</th>
<th>Number of paths</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$2^1 + 1 = 3$</td>
</tr>
<tr>
<td>2</td>
<td>$2^2 + 1 = 5$</td>
</tr>
<tr>
<td>3</td>
<td>$2^3 + 1 = 9$</td>
</tr>
<tr>
<td>10</td>
<td>1.025</td>
</tr>
<tr>
<td>20</td>
<td>1.048.577</td>
</tr>
</tbody>
</table>

2. Path
# Limits of Testing

Branching and dynamic binding result in a very large number of unique execution sequences. Simple iteration increases the number of possible sequences to astronomical proportions.

<table>
<thead>
<tr>
<th>Number of iterations</th>
<th>Number of paths</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$2^1 + 1 = 3$</td>
</tr>
<tr>
<td><strong>2</strong></td>
<td><strong>$2^2 + 1 = 5$</strong></td>
</tr>
<tr>
<td>3</td>
<td>$2^3 + 1 = 9$</td>
</tr>
<tr>
<td>10</td>
<td>1.025</td>
</tr>
<tr>
<td>20</td>
<td>1.048.577</td>
</tr>
</tbody>
</table>

3. Path
Limits of Testing

Branching and dynamic binding result in a very large number of unique execution sequences. Simple iteration increases the number of possible sequences to astronomical proportions.

<table>
<thead>
<tr>
<th>Number of iterations</th>
<th>Number of paths</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$2^1 + 1 = 3$</td>
</tr>
<tr>
<td>2</td>
<td>$2^2 + 1 = 5$</td>
</tr>
<tr>
<td>3</td>
<td>$2^3 + 1 = 9$</td>
</tr>
<tr>
<td>10</td>
<td>1.025</td>
</tr>
<tr>
<td>20</td>
<td>1.048.577</td>
</tr>
</tbody>
</table>

4. Path
Limits of Testing

Branching and dynamic binding result in a very large number of unique execution sequences. Simple iteration increases the number of possible sequences to astronomical proportions.

<table>
<thead>
<tr>
<th>Number of iterations</th>
<th>Number of paths</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$2^1 + 1 = 3$</td>
</tr>
<tr>
<td>2</td>
<td>$2^2 + 1 = 5$</td>
</tr>
<tr>
<td>3</td>
<td>$2^3 + 1 = 9$</td>
</tr>
<tr>
<td>10</td>
<td>1.025</td>
</tr>
<tr>
<td>20</td>
<td>1.048.577</td>
</tr>
</tbody>
</table>
The ability of code to hide faults from a test suite is called its fault sensitivity.

Coincidental correctness is obtained when buggy code can produce correct results for some inputs.
E.g. assuming that the correct code would be:
\[ x = x + x \]
but you wrote
\[ x = x \times x \]
If \( x = 2 \) or \( x = 0 \) is tested the code hides the bug: it produces a correct result from buggy code. However, this bug is easily identified.
Implementing Tests

• A Very First Glimpse
static long process(String[] args) throws IllegalArgumentException {

    Stack values = new Stack();
    for (int i = 0; i < args.length; i++) {
        String arg = args[i];
        try {
            long value = Long.parseLong(arg);
            values.push(value);
        } catch (NumberFormatException nfe) {
            // there is no method to test if a string is
            if (values.size() > 1) {
                long r = values.pop();
                long l = values.pop();
                if (arg.equals("+")) {
                    values.push(l + r);
                    continue;
                }
                if (arg.equals("*")) {
                    values.push(l * r);
                    continue;
                }
            }
            throw new IllegalArgumentException("Too few operands or operator unknown."));
        }
    }
    if (values.size() == 1) return values.pop();
    else throw new IllegalArgumentException("Too few (0) or too many (>1) operands.");
}
## Description

<table>
<thead>
<tr>
<th>Description</th>
<th>Input</th>
<th>Expected Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test calculation of the correct result</td>
<td>&quot;4&quot;, &quot;5&quot;, &quot;+&quot;, &quot;7&quot;, &quot;*&quot;</td>
<td>63</td>
</tr>
<tr>
<td>Test that too few operands leads to the corresponding exception</td>
<td>&quot;4&quot;, &quot;5&quot;, &quot;+&quot;, &quot;*&quot;</td>
<td>Exception: &quot;Too few operands or operator unknown.&quot;</td>
</tr>
<tr>
<td>Test that an illegal operator / operand throws the corresponding exception</td>
<td>&quot;4&quot;, &quot;5327h662h&quot;, &quot;*&quot;</td>
<td>Exception: &quot;Too few operands or operator unknown.&quot;</td>
</tr>
<tr>
<td>Test that an expression throws the corresponding exception</td>
<td>{}</td>
<td>Exception: &quot;Too few (0) or too many ((&gt;1)) operands left.&quot;</td>
</tr>
<tr>
<td>Test that too few operates leads to the corresponding exception</td>
<td>&quot;4&quot;, &quot;5&quot;</td>
<td>Exception: &quot;Too few (0) or too many ((&gt;1)) operands left.&quot;</td>
</tr>
<tr>
<td>Description</td>
<td>Input</td>
<td>Expected Output</td>
</tr>
<tr>
<td>--------------------------------------------------</td>
<td>------------------------</td>
<td>------------------------------------------------------</td>
</tr>
<tr>
<td>Test calculation of the correct result</td>
<td>&quot;4&quot;, &quot;5&quot;, &quot;+&quot;, &quot;7&quot;, &quot;*&quot;</td>
<td>63</td>
</tr>
<tr>
<td>Test that too few operands leads to the corresponding exception</td>
<td>&quot;4&quot;, &quot;5&quot;, &quot;+&quot;, &quot;*&quot;</td>
<td>Exception: &quot;Too few operands or operator unknown.&quot;</td>
</tr>
<tr>
<td>Test that an illegal operator operand throws the corresponding exception</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Test that an expression throws the corresponding exception</td>
<td>{}</td>
<td>Exception: &quot;Too few (0) or too many (&gt;1) operands left.&quot;</td>
</tr>
<tr>
<td>Test that too few operates leads to the corresponding exception</td>
<td>&quot;4&quot;, &quot;5&quot;</td>
<td>Exception: &quot;Too few (0) or too many (&gt;1) operands left.&quot;</td>
</tr>
</tbody>
</table>
static long process(java.lang.String[] args)

0
stack = new demo.SimpleCalculator.Stack
stack.<init>()

1
p14 = Φ(0←0, p65←11)
p17 = args.length
if(p17 > p14)

2
p20 = args[p14]

3
long p23 = java.lang.Long.parseLong(p20)
stack.push(p23)

5
java.lang.NumberFormatException
int p30 = stack.size()
if(1 >= p30)

8
long p40 = stack.pop();
long p43 = stack.pop();
p47 = p20.equals("+");
if(p47)

9
p50 = p20.equals("=");
if(p50)

10
long p54 = p43 * p40
stack.push(p54)

12
long p58 = p43 + p40
stack.push(p58)

13
p58 = p43 + p40
stack.push(p58)

14
p70 = stack.size();
if(p70 >= 1)

16
p72 = new java.lang.IllegalArgumentException
p72.<init>_Release()
throw p72

17
p76 = stack.pop();
return p76

18
Exit

19
p54 = p43 * p40
stack.push(p54)

20
p58 = p43 + p40
stack.push(p58)

21
p23 = java.lang.Long.parseLong(p20)
stack.push(p23)

22
java.lang.NumberFormatException
int p30 = stack.size();
if(1 >= p30)

23
p47 = p20.equals("+");
if(p47)

24
p50 = p20.equals("=");
if(p50)

25
long p40 = stack.pop();
long p43 = stack.pop();
p47 = p20.equals("+");
if(p47)

26
p50 = p20.equals("=");
if(p50)

27
long p54 = p43 * p40
stack.push(p54)

28
long p58 = p43 + p40
stack.push(p58)

29
p70 = stack.size();
if(p70 >= 1)

30
p72 = new java.lang.IllegalArgumentException
p72.<init>_Release()
throw p72

31
p76 = stack.pop();
return p76

32
Exit
static long process(java.lang.String[] args)
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.fail;
import java.util.Arrays;
import org.junit.Test;

public class SimpleCalculatorTest extends ...
{

    public void testProcess()
    {
        try {
            SimpleCalculator.process(new String[0]);
        } catch (IllegalArgumentException iae) {
            assertEquals("Too few (0) or too many (>1) operands.", iae.getMessage());
        }
    }
}
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.fail;
import java.util.Arrays;
import org.junit.Test;

public class SimpleCalculatorTest {

    @Test
    public void testProcess() {
        String[] term = new String[] {
            "4", "5", "+", "7", "*"
        };
        long result = SimpleCalculator.process(term);
        assertEquals(Arrays.toString(term), 63, result);
    }
}

Writing a Test Case using JUnit (4)
Writing a Test Case using JUnit (4) - Testing Exception Handling

```java
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.fail;
import java.util.Arrays;
import org.junit.Test;

public class SimpleCalculatorTest {

    @Test(expected=IllegalArgumentException.class)
    public void testProcess() {
        SimpleCalculator.process(new String[0]);
    }
}
```
Alternative Frameworks for Writing Tests

TestNG

// This method will provide data to any test method
// that declares that its Data Provider is named "provider1".
@DataProvider(name = "provider1")
public Object[][] createData1() {
    return new Object[][] {
        { "Cedric", new Integer(36) },
        { "Anne", new Integer(37) }
    };
}

// This test method declares that its data should be
// supplied by the Data Provider named "provider1".
@Test(dataProvider = "provider1")
public void verifyData1(String n1, Integer n2) {
    System.out.println(n1 + " " + n2);
import static org.hamcrest.MatcherAssert.assertThat;
import static org.hamcrest.Matchers.*;

import junit.framework.TestCase;

public class BiscuitTest extends TestCase {
    public void testEquals() {
        Biscuit theBiscuit = new Biscuit("Ginger");
        Biscuit myBiscuit = new Biscuit("Ginger");
        assertThat(theBiscuit, equalTo(myBiscuit));
    }
}
ScalaTest
(Can also be used for testing Java.)

```scala
class DefaultIntegerRangesTest extends FunSpec with Matchers with ParallelTestExecution {
  describe("IntegerRange values") {
    describe("the behavior of irem") {
      it("AnIntegerValue % AnIntegerValue => AnIntegerValue + Exception") {
        val v1 = AnIntegerValue()
        val v2 = AnIntegerValue()

        val result = irem(-1, v1, v2)
        result.result shouldBe an[AnIntegerValue]
        result.exceptions match {
          case SObjectValue(ObjectType.ArithmeticException) => /*OK*/
          case v => fail(s"expected ArithmeticException; found $v")
        }
      }
    }
  }
}
```

small concise tests ("atomic tests")

very good support for Pattern Matching
Behavior-Driven Development

The goal is that developers define the behavioral intent of the system that they are developing.

http://behaviour-driven.org/

---

import org.specs.runner._
import org.specs._

object SimpleCalculatorSpec extends Specification {

  "The Simple Calculator" should {
    "return the value 36 for the input {"6","6","*"}" in {
      SimpleCalculator.process(Array("6","6","*")) must_== 36
    }
  }

}

Using ScalaSpec 1.5: http://code.google.com/p/specs/
(Method-) Stub

- A stub is a partial, temporary implementation of a component (e.g., a placeholder for an incomplete component)
- Stubs are often required to simulate complex systems; to make parts of complex systems testable in isolation

An alternative is to use a Mock object that mimics the original object in its behavior and facilitates testing.
Testing comprises the efforts to find defects.

Debugging is the process of locating and correcting defects.

(Hence, debugging is not testing, and testing is not debugging.)
The goal of this lecture is to enable you to systematically carry out small(er) software projects that produce quality software.

- Testing has to be done systematically; exhaustive testing is not possible.
- Test coverage models help you to assess the quality of your test suite; however, “just” satisfying a test coverage goal is usually by no means sufficient.
- Do take an “external” perspective when you develop your test suite.
The goal of this lecture is to enable you to systematically carry out small(er) commercial or open-source projects.
A Tester’s Courage

The Director of a software company proudly announced that a flight software developed by the company was installed in an airplane and the airline was offering free first flights to the members of the company. “Who are interested?” the Director asked. Nobody came forward. Finally, one person volunteered. The brave Software Tester stated, ‘I will do it. I know that the airplane will not be able to take off.’