System Testing

Steven J Zeil

April 9, 2013

Contents

1 Test Coverage 3
  1.1 Coverage Measures 3
System Testing

1.1.1 Black-Box Testing ................................................................. 3
1.1.2 White-Box Testing ................................................................. 7
1.2 C/C++ - gcov ................................................................. 26
1.3 Java ................................................................. 50

2 Oracles ................................................................. 70
2.1 expect ................................................................. 71
2.2 *Unit ................................................................. 71
2.3 GUI systems ................................................................. 73
2.4 Web systems ................................................................. 76
2.5 Selenium ................................................................. 78
1 Test Coverage

1.1 Coverage Measures

1.1.1 Black-Box Testing

Black-Box Testing

Black-box (a.k.a. specification-based) testing chooses tests without consulting the implementation.

- Equivalence partitioning
- Boundary-value testing
- Special-values testing
System Testing

**Equivalence Partitioning**
(a.k.a *functional testing*)

- Attempt to choose test data illustrating each distinct behavior or each distinct class of inputs and outputs at least once.
  - e.g., each kind of transaction, each kind of report
- Can be driven by function points.

**Boundary-Values Testing**

Choose data at the boundaries of a functional testing class or of the overall input domain.
System Testing

- check amount = 0
- check amount $\geq 1,000,000$
- transaction date = day before bank was founded
- transaction date 100 years in future
- name string empty
- name string one less than full
- name string full
- name string overfull
Special-Values Testing
Choose data reflecting “special” or troublesome cases.
Examples include choosing for

• each numeric input
  – negative,
  – zero, and
  – positive values,

• each string input
  – empty
  – entirely blank strings,
1.1.2 White-Box Testing

White-Box Testing

*White-Box* (a.k.a. *Implementation-based* testing) uses information from the implementation to choose tests.

- Structural Testing (a.k.a., “path testing” (not per your text))
  Designate a set of paths through the program that must be exercised during testing.
  - Statement Coverage
System Testing

- Branch Coverage
- Cyclomatic coverage ("independent path testing")
- Data-flow Coverage

• Mutation testing

Statement Coverage
Require that every statement in the code be executed at least once during testing.

• Needs software tools to monitor this requirement for you.
  - e.g., gcov in Unix for C, C++
Statement Coverage Example

```cpp
int main() {
    int x, y;
    cin >> x >> y;
    while (x > y) {
        if (x > 0)
            cout << x;
        x = f(x, y);
    }
    cout << x;
    return 0;
}
```

What kinds of tests are required for statement coverage?

.........................

Branch Coverage

Requires that every "branch" in the flowchart be tested at least once
System Testing

- Equivalent to saying that each conditional stmt must be tested as both true and false

- Branch coverage implies Statement Coverage, but not vice versa

```c
if (X < 0)
    X = -X;
Y = sqrt(X);
```

Branch Coverage Example

```c
cin >> x >> y;
while (x > y)
{
```

CS795 10
System Testing

```cpp
if (x > 0)
    cout << x;
    x = f(x, y);
}
cout << x;
```

What kinds of tests are required for branch coverage?

Variations on Branch Coverage

- **Path coverage** seeks to cover each path from start to finish through the program.
  - Infeasible (why?)
- Loop coverage: various rules such as
A loop is covered if, in at least one test, the body was executed 0 times, and if in some test the body was executed exactly once, and if in some test the body was executed more than once.

Multi-Condition Coverage
a.k.a., Condition coverage

- Various approaches to coping with boolean expressions, particularly short-circuited ones.
- Goal: given a boolean expression $a \oplus b$, where $\oplus$ could be &, &&, |, etc., need at least one test where
– $a$ is true and, had it been false, the value of $a \oplus b$ would change
– $a$ is false and, had it been true, the value of $a \oplus b$ would change
– $b$ is true and, had it been false, the value of $a \oplus b$ would change
– $b$ is false and, had it been true, the value of $a \oplus b$ would change

• For example, for the expression $a \& b$, we would need the combinations

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>$b$</td>
</tr>
<tr>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
</tr>
</tbody>
</table>
Cyclomatic Coverage
System Testing

(a.k.a “independent path coverage”, “path testing”)

• The latter term (used in your text) should be discouraged as it is both vague and means something entirely different to most of the testing community

• Each independent path must be tested
  – An independent path is one that includes a branch not previously taken.

Cyclomatic Example
What are the independent paths?

One set:

1, 2, 3, 4, 12, 13
1, 2, 3, 5, 6, 11, 2, 12, 13
1, 2, 3, 5, 7, 8, 10, 11, 2, 12, 13
1, 2, 3, 5, 7, 9, 10, 11, 2, 12, 13

.........................

Cyclomatic Complexity
The number of independent paths in a program can be discovered by computing the cyclomatic complexity (McCabe, 1976) …

\[ CC(G) = \text{Number(edges)} - \text{Number(nodes)} + 1 \]

- This is a popular metric for module complexity.
- Actually pretty trivial: for structured programs with only binary decision constructs, equals number of conditional statements + 1
- relation to testing is dubious
  - simply branch coverage hidden behind smoke and mirrors
Issues

- Sets of independent paths are not unique, nor is their size:
1,2,3,5,6,11,
2,3,5,7,8,
10,11,2,3,
5,7,9,10,11,
2,12,13

1,2,3,4,12,13

Data-Flow Coverage
System Testing

Attempts to test significant combinations of branches.

- Any stmt \(i\) where a variable \(X\) may be assigned a new value is called a definition of \(X\) at \(i\): \(\text{def}(X,i)\)

- Any stmt \(i\) where a variable \(X\) may be used/retrieved is called a reference or use of \(X\) at \(i\): \(\text{ref}(X,i)\)

Def-Clear Paths

- A path from stmt \(i\) to stmt \(j\) is def-clear with respect to \(X\) if it contains no definitions of \(X\) except possibly at the beginning (\(i\)) and end (\(j\))
all-defs

The *all-defs* criterion requires that each definition `def(X, i)` be tested some def-clear path to some reference `ref(X, j)`.

```cpp
1: cin >> x >> y;  // d(x,1) d(y,1)
2: while (x > y)  // r(x,2), r(y,2)
3: {
4:   if (x > 0)  // r(x,4)
5:     cout << x;  // r(x,5)
6:   x = f(x, y);  // r(x,6), r(y,6), d(x,6)
7: }
8: cout << x;  // r(x,8)
```

What kinds of tests are required for all-defs coverage?

.................................
all-uses

The *all-uses* criterion requires that each pair \((\text{def}(X,i), \text{ref}(X,j))\) be tested using some def-clear path from \(i\) to \(j\).

```c
1: cin >> x >> y;  // d(x,1)  d(y,1)
2: while (x > y)   // r(x,2),  r(y,2)
3: {
4:     if (x > 0)  // r(x,4)
5:         cout << x;  // r(x,5)
6:     x = f(x, y);  // r(x,6),  r(y,6),  d(x,6)
7: }              // r(x,8)
8: cout << x;      // r(x,8)
```

What kinds of tests are required for all-uses coverage?

.................................

CS795 22
System Testing

Mutation Testing

Given a program $P$,

- Form a set of mutant programs that differ from $P$ by some single change

- These changes (called mutation operators) include:
  
  - exchanging one variable name by another
  - altering a numeric constant by some small amount
  - exchanging one arithmetic operator by another
  - exchanging one relational operator by another
  - deleting an entire statement
  - replacing an entire statement by an `abort()` call
Mutation Testing (cont.)

- Run $P$ and each mutant $P_i$ on a previously chosen set of tests
- Compare the output of each $P_i$ to that of $P$
  - If the outputs differ on any test, $P_i$ is *killed* and removed from the set of mutant programs
  - If the outputs are the same on all tests, $P_i$ is still considered *alive*.

Mutation Testing (cont.)

A set of test data is considered *inadequate* if it cannot distinguish between the program as written ($P$) and programs that differ from it by only a simple change.
System Testing

• So if any mutants are still alive after running a set of tests, we augment the tests until we can kill all the mutants.

.................................

Mutation Testing Problems

• Even simple programs yield tens of thousands of mutants. Executing these is time-consuming.
  – But most are killed on first few tests
  – And the process *is* automated
Some mutants are actually equivalent to the original program:

\[
\begin{align*}
X &= Y; \\
X &= Y; \\
\text{if } (X > 0) &\quad \text{if } (Y > 0) \\
\vdots &\quad \vdots
\end{align*}
\]

- Identifying these can be difficult (and cannot be automated)

1.2 C/C++ - gcov

Monitoring Statement Coverage with gcov

- coverage tool includes with the GNU compiler suite (gcc, g++, etc.)

- As an example, look at testing the three search functions in CS795
```cpp
#ifndef ARRAYUTILS_H
#define ARRAYUTILS_H

// Add to the end
// - Assumes that we have a separate integer (size) indicating how many elements are in the array
// - and that the "true" size of the array is at least one larger than the current value of that counter

template<typename T>
void addToEnd (T* array, int& size, T value)
{
    array[size] = value;
}
```

System Testing

```cpp
template <typename T>
void addElement (T* array, int& size, int index, T value)
```

// Add value into array[index], shifting all elements already in
// index..size-1 up one, to make room.
// - Assumes that we have a separate integer (size) indicating how
//   many elements are in the array
// - and that the "true" size of the array is at least one larger
//   than the current value of that counter

++size;
}
```
// Make room for the insertion
int toBeMoved = size - 1;
while (toBeMoved >= index) {
    array[toBeMoved+1] = array[toBeMoved];
    --toBeMoved;
}
// Insert the new value
array[index] = value;
++size;

// Assume the elements of the array are already in order
// Find the position where value could be added to keep everything in order, and insert it there.
// Return the position where it was inserted
// - Assumes that we have a separate integer (size) indicating how many elements are in the array
// - and that the "true" size of the array is at least one larger than the current value of that counter

template <typename T>
int addInOrder (T* array, int& size, T value)
{
    // Make room for the insertion
    int toBeMoved = size - 1;
    while (toBeMoved >= 0 && value < array[toBeMoved]) {
        // Move elements up to make room for the insertion
        array[toBeMoved + 1] = array[toBeMoved];
        toBeMoved--;
    }
    array[toBeMoved + 1] = value;
    size++;
    return toBeMoved + 1;
}
System Testing

```cpp
array[toBeMoved+1] = array[toBeMoved];
--toBeMoved;
}

// Insert the new value
array[toBeMoved+1] = value;
++size;
return toBeMoved+1;
}

// Search an array for a given value, returning the index where
// found or -1 if not found.
template <typename T>
int seqSearch(const T list[], int listLength, T searchItem)
```
{  
    int loc;
    
    for (loc = 0; loc < listLength; loc++)
        if (list[loc] == searchItem)
            return loc;
    
    return -1;
}

// Search an ordered array for a given value, returning the index where
// found or -1 if not found.

template<typename T>
```c
int seqOrderedSearch(const T list[], int listLength, T searchItem)
{
    int loc = 0;

    while (loc < listLength && list[loc] < searchItem)
    {
        ++loc;
    }

    if (loc < listLength && list[loc] == searchItem)
        return loc;
    else
        return -1;
}
```
// Removes an element from the indicated position in the array, moving
// all elements in higher positions down one to fill in the gap.

template <typename T>
void removeElement (T* array, int& size, int index) {
    int toBeMoved = index + 1;
    while (toBeMoved < size) {
        array[toBeMoved] = array[toBeMoved+1];
        ++toBeMoved;
    }
    --size;
}
// Search an ordered array for a given value, returning the index where found or -1 if not found.

template <typename T>
int binarySearch(const T list[], int listLength, T searchItem)
{
    int first = 0;
    int last = listLength - 1;
    int mid;

    bool found = false;

    while (first <= last && !found)
{ 
    mid = (first + last) / 2;

    if (list[mid] == searchItem)
        found = true;
    else
        if (searchItem < list[mid])
            last = mid - 1;
        else
            first = mid + 1;
    }

    if (found)
        return mid;
else
    return -1;
}
#endif

- with test driver

#include <cassert>
#include <iostream>
#include <sstream>
#include <string>

#include "arrayUtils.h"

using namespace std;

// Unit test driver for array search functions
```cpp
int main(int argc, char** argv)
{
    // Repeatedly reads tests from cin
    // Each test consists of a line containing one or more words. The
    // first word is one that we want to search for. The remaining
    // words are placed into an array and represent the collection
    // we will search through.

    string line;
    getline (cin, line);
    while (cin)
    {
        istringstream in (line);
        cout << line << endl;
    }
}
```
string toSearchFor;
    in >> toSearchFor;
    int nWords = 0;
    string words[100];
    while (in >> words[nWords])
        ++nWords;

    cout << seqSearch (words, nWords, toSearchFor)  
        << " "  
        << seqOrderedSearch (words, nWords, toSearchFor)  
        << " "  
        << binarySearch (words, nWords, toSearchFor)  
        << endl;
getline (cin, line);
}

return 0;
}

, which reads data from a text stream (e.g., standard in), uses that data to construct arrays, and invokes each function on those arrays, printing the results of each.

.................................

Compiling for gcov Statement Coverage

- To use gcov, we compile with special options
- `fprofile-arcs -ftest-coverage`

- When the code has been compiled, in addition to the usual files there will be several files with endings like `.gcno`

  - These hold data on where the statements and branches in our code are.

Running Tests with `gcov`

- Run your tests normally.

- As you test, a `*.gcda` file will accumulate
System Testing

Viewing Your Report

• Run gcov `mainProgram`

  – The immediate output will be a report on the percentages of statements covered in each source code file.
  – Also creates a `*.gcov` detailed report for each source code file. e.g.,

    ........................................

Sample Statement Coverage Report

−: 69:template <typename T>
−: 70:int seqSearch(const T list[], int listLength, T searchItem)
−: 71:{
System Testing

1: 72: int loc;
−: 73:
2: 74: for (loc = 0; loc < listLength; loc++)
2: 75: if (list[loc] == searchItem)
1: 76: return loc;
−: 77:
####: 78: return -1;
−: 79:}

• Report lists number of times each statement has been executed
  – Lists ### if a statement has never been executed


CS795 44
Monitoring Branch Coverage with gcov

gcov can report on branches taken.

• Just add options to the gcov command:
  
  - gcov -b -c mainProgram

Reading gcov Branch Info

• gcov reports
  
  - Number of times each function call successfully returned
System Testing

– # of times a branch was executed (i.e., how many times the branch condition was evaluated)
– and # times each branch was taken
  * For branch coverage, this is the relevant figure

But What is a “Branch”?

• A "branch" is anything that causes the code to not continue on in straight-line fashion
  – Branch listed right after an "if" is the "branch" that jumps around the "then" part to go to the "else" part.
– && and || operators introduce their own branches
– Other branches may be hidden
  * Contributed by calls to inline functions
  * Or just a branch generated by the compiler’s code generator
• In practice, this can be very hard to interpret

Example: gcov Branch Coverage report

```cpp
84: template <typename T>
85: int seqOrderedSearch(const T list[], int listLength, T searchItem)
86: {
87:   int loc = 0;
```
System Testing

```
while (loc < listLength && list[loc] < searchItem)
branch 0 taken 0
call 1 returns 1
branch 2 taken 0
branch 3 taken 1

{ 
  ++loc;

branch 0 never executed

}

if (loc < listLength && list[loc] == searchItem)
branch 0 taken 0
call 1 returns 1
branch 2 taken 0

return loc;
```
• Report is organized by *basic blocks*, straight-line sequences of code terminated by a branch or a call

• Hard to map to specific source code constructs
  – lowest-numbered branch is often the leftmost condition
  – Fact of life that compilers insert branches and calls that are often invisible to us
1.3 Java

Java Coverage Tools

- Clover
- JaCoCo
  - Part of the EclEmma project (Eclipse plugin for Emma)
  - Emma, an older coverage tool, now replaced by JaCoCo

Clover

- Commercial product, currently free for open-source projects
System Testing

- integrates with Ant, Maven
- lots of reporting features

• Works in “traditional” coverage tool fashion
  - Requires a “fork” of the build process to build a monitoring version
  - Injects monitors into compiled code

• Test optimization: can re-run only those tests that covered changed code

............................

JaCoCo

• line and branch coverage
System Testing

• Instrumentation is done on the fly
  – An “agent” monitors execution of normally compiled bytecode
    * No special build required

• Supports full Java 7

• Works with Maven & Ant
  – In Ant, wrap normal <java> and <junit> tasks inside a <jacoco:coverage> element
Example: JaCoCo in Ant

Working with our Code Annotation project, add a dependency on the JaCoCo library:

```xml
<ivy-module version="2.0">
  <info organisation="edu.odu.cs" module="codeAnnotation" revision="1.0"/>
  <publications>
    <artifact name="codeAnnotation" type="jar" ext="jar"/>
    <artifact name="codeAnnotation-src" type="source" ext="zip"/>
  </publications>
  <dependencies>
    <dependency org="de.jflex" name="jflex" rev="1.4.3"/>
    <dependency org="junit" name="junit" rev="4.10"/>
    <dependency org="org.jacoco" name="org.jacoco.ant" rev="latest.integration"/>
  </dependencies>
</ivy-module>
```
</dependencies>
</ivy-module>

Example: JaCoCo in Ant (cont.)

<project name="codeAnnotation" basedir="." default="build"
    xmlns:ivy="antlib:org.apache.ivy.ant"
    xmlns:jacoco="antlib:org.jacoco.ant"
>

    <record name="ant.log" action="start" append="false" />

</project>
<path id="testCompilationPath">
    <fileset dir="lib" includes="*.jar"/>
    <pathelement path="target/classes"/>
</path>

<path id="testExecutionPath">
    <fileset dir="lib" includes="*.jar"/>
    <pathelement path="target/classes"/>
    <pathelement path="target/test-classes"/>
</path>

<property name="ivy.install.version" value="2.3.0"/>
<property name="jsch.install.version" value="0.1.49"/>
<property name="ivy.jar.dir" value="${basedir}/ivy"/>
<property name="ivy.jar.file" value="${ivy.jar.dir}/ivy.jar"/>
<property name="jsch.jar.file" value="${ivy.jar.dir}/jsch.jar"/>
<property name="build.dir" value="build"/>
<property name="src.dir" value="src"/>

<target name="download-ivy" unless="skip.download">
  <mkdir dir="${ivy.jar.dir}"/>
  <echo message="installing ivy..."/>
  <get src="http://repo1.maven.org/maven2/org/apache/ivy/ivy/${ivy.install.version}/ivy-${ivy.install.version}.jar" dest="${ivy.jar.file}" usetimestamp="true"/>
</target>

<target name="download-jsch" unless="skip.download"/>
<mkdir dir="${ivy.jar.dir}"/>
<echo message="installing jsch..."/>
<get src="http://repo1.maven.org/maven2/com/jcraft/jsch/${jsch.install.version}/jsch-${jsch.install.version}.jar" dest="${jsch.jar.file}" usetimestamp="true"/>
</target>

<target name="install-jsch" depends="download-jsch">
</target>

<target name="install-ivy" depends="download-ivy" description="--> install ivy">
<path id="ivy.lib.path">
  <fileset dir="${ivy.jar.dir}" includes="*.jar"/>
</path>
</target>
<taskdef resource="org/apache/ivy/ant/antlib.xml"
    uri="antlib:org.apache.ivy.ant" classpathref="ivy.lib.path"/>
</target>

<target name="resolve-ivy" depends="install-ivy,install-jsch" description="">
    <ivy:retrieve/>
    <echo>ivy.default.ivy.user.dir is ${ivy.default.ivy.user.dir}</echo>
    <taskdef uri="antlib:org.jacoco.ant" resource="org/jacoco/ant/antlib.xml">
        <classpath refid="testExecutionPath"/>
    </taskdef>
    <taskdef classname="JFlex.anttask.JFlexTask" name="jflex">
        <classpath refid="testExecutionPath"/>
    </taskdef>
</target>
<target name="generateSource" depends="resolve-ivy">
    <mkdir dir="target/gen/java"/>
    <jflex file="src/main/jflex/code2html.flex"
        destdir="target/gen/java"/>
    <jflex file="src/main/jflex/code2tex.flex"
        destdir="target/gen/java"/>
    <jflex file="src/main/jflex/list2html.flex"
        destdir="target/gen/java"/>
    <jflex file="src/main/jflex/list2tex.flex"
        destdir="target/gen/java"/>
</target>
<target name="compile" depends="generateSource">
    <mkdir dir="target/classes"/>
    <javac srcdir="target/gen/java" destdir="target/classes" debug="true"
        source="1.6" includeantruntime="false"/>
    <javac srcdir="src/main/java" destdir="target/classes" debug="true"
        source="1.6" includeantruntime="false"/>
</target>

<target name="compile-tests" depends="compile">
    <mkdir dir="target/test-classes"/>
    <javac srcdir="src/test/java" destdir="target/test-classes" debug="true"
        source="1.6" includeantruntime="false">
        <classpath refid="testCompilationPath"/>
    </javac>
</target>
<target name="test" depends="compile-tests">
  <mkdir dir="target/test-results/details"/>
  <jacoco:coverage destfile="target/jacoco.exec">
    <junit printsummary="yes"
      haltonfailure="yes" fork="yes"
    >
    <classpath refid="testExecutionPath"/>
    <formatter type="xml"/>
    <batchtest todir="target/test-results/details">
      <fileset dir="target/test-classes">
        <include name="**/*Test*.class"/>
      </fileset>
    </batchtest>
  </jacoco:coverage>
</target>
<target name="coverageReport" depends="test">
    <jacoco:report>
        <executiondata>
            <fileset dir="target/test-results/details">
                <include name="TEST-*.xml"/>
            </fileset>
            <report format="frames" todir="target/test-results/html"/>
        </executiondata>
    </jacoco:report>
</target>
<file file="target/jacoco.exec"/>
</executiondata>

<structure name="Code Annotation Project">
  <classfiles>
    <fileset dir="target/classes"/>
    <fileset dir="target/test-classes"/>
  </classfiles>
  <sourcefiles encoding="UTF-8">
    <fileset dir="src/main/java"/>
    <fileset dir="src/test/java"/>
    <fileset dir="target/gen/java"/>
  </sourcefiles>
</structure>
System Testing

```xml
<html destdir="target/coverageReport"/>
</jacoco:report>
</target>

<target name="build" depends="coverageReport">
  <jar destfile="codeAnnotation.jar" basedir="target/classes">
    <manifest>
      <attribute name="Main-Class" value="edu.odu.cs.code2html.Code2HTML"/>
    </manifest>
  </jar>
  <zip destfile="target/codeAnnotation-src.zip">
```

CS795  64
<fileset dir=".">
  <include name="*.xml"/>
  <include name="test.cpp"/>
  <include name="*.css.cpp"/>
  <include name="src/**/*"/>
  <exclude name="**/*~"/>
  <exclude name="target/**/*"/>
</fileset>
</zip>
</target>

<target name="publish" depends="build">
  <ivy:retrieve/>
  <ivy:publish resolver="Forge350Publish"
<target name="clean">
    <delete dir="target"/>
</target>

<target name="cleaner" depends="clean">
    <delete dir="lib"/>
</target>
Once the dependencies are resolved, we can activate the JaCoCo tasks.

Note that there is no change at all in compilation

And minimal change to execution

- Test execution must have fork="true" because
System Testing

* agent needs to be attached to the running JVM
* (which is already running **ant**) – In practice, I might coverage data collection a separate target

 Preparation of reports starts here

  ④ Must match destination given when running tests
  ⑤ This describes the class and source code file locations
  ⑥ Choose report format and location

  
  Example: JaCoCo Report
System Testing

• Report
  – Notice that even JFlex-generated code gets measured and included in report
    * Though the annotated listings are missing for some reason.

EclEmma
  Eclipse plugin for coverage tools (JaCoCo)
  • Adds a new launch mode, *Coverage mode*, for running programs similar to normal
    “run” and “debug” modes

• Reports include
System Testing

- Summary Coverage View
- Can highlight coverage in Eclipse code editors as colored annotations

2 Oracles

Oracles
A testing oracle is the process, person, and/or program that determines if test output is correct
2.1 expect

expect
Covered previously, **expect** is a shell for testing interactive programs.

- an extension of **TCL** (a portable shell script).
- Largely confined to text streams as input/output

2.2 *Unit

*Unit
Can we use *Unit-style frameworks as oracles at the system test level?
System Testing

• The very question is heresy to many *Unit advocates
  – Particularly runs counter to the goals of the various Mock Objects projects

• But, why not?
  – Such tests do not (should not) be at the expense of having done earlier “proper” unit testing.
  – Particularly in Java, MyClass.main(String[]) can be called just like any other function
    * And System.in/varname cin and System.out/cout can be rerouted to/from files or internal strings
  – Major limitation is the accessibility of system inputs & outputs.
    * GUIs, data bases, etc.
2.3 GUI systems

GUI testing

- Scripting or record/playback: playing back input events for
  - convenience & efficiency
  - consistent reproducibility

- Capture of results
  - Can occur at different levels
    * event/message level
* graphics level

Some Open Alternatives

- Marathon - free in limited version
- Jemmy

Marathon
For Java GUIs
System Testing

- Recorder captures AWT/swing events as JRuby scripts
- Scripts can then be edited to alter inputs, add assertions, etc.

```ruby
def test
  $java_recorded_version = "1.6.0_24"
  with_window("Simple Widgets") {
    select("First Name", "Jalian Systems")
    select("Password", "Secret")
    assert_p("First Name", "Text", "Jalian Systems")
  }
end
```

..............................
System Testing

Jemmy
Also for Java GUIs

• Tests scripted as Java
• Integrates with JUnit
  – Example

2.4 Web systems

Web systems

• A subproblem of GUI testing
System Testing

- Simpler because input structure more contrained
- Output detail level is fixed (http: events)

Some Open Alternatives

- Selenium
- antEater
- Watir
2.5 **Selenium**

**Selenium**

- Browser automation (SeleniumIDE - Firefox add-on)
  - Record & playback
  - Or scripted (Selenium Webdriver)
    * Firefox, IE, Safari, Opera, Chrome

............................

**Selenium Scripting**
System Testing

• Actions do things to elements.
  E.g., click buttons, select options

• Accessors examine the application state

• Assertions validate the state
  Each assertion has 3 modes
  – assert: failure aborts the test
  – verify: test continues, but failure is logged
  – waitFor: conditions that may be true immediately or may become true within a specified time interval
Selenese

A typical scripting statement has the form

**Syntax**

```
command parameter1 [parameter2]
```

Parameters can be

- locators for finding a UI element within a page (xpath)
- text patterns
- variable names
A Sample Selenium Script

```html
<table>
  <tr>
    <td>open</td>
    <td>http://mySite.com/downloads/</td>
  </tr>
  <tr>
    <td>assertTitle</td>
    <td>Downloads</td>
  </tr>
  <tr>
    <td>verifyText</td>
    <td>//h2</td>
    <td>Terms and Conditions</td>
  </tr>
  <tr>
    <td>clickAndWait</td>
    <td>//input[@value="I agree"]</td>
  </tr>
  <tr>
    <td>assertTitle</td>
    <td>Product Selection</td>
  </tr>
</table>
```

That's right – it's an HTML table:

<table>
<thead>
<tr>
<th><strong>open</strong></th>
<th><a href="http://mySite.com/downloads/">http://mySite.com/downloads/</a></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>assertTitle</strong></td>
<td>Downloads</td>
</tr>
<tr>
<td><strong>verifyText</strong></td>
<td>//h2</td>
</tr>
<tr>
<td><strong>clickAndWait</strong></td>
<td>//input[@value=&quot;I agree&quot;]</td>
</tr>
<tr>
<td><strong>assertTitle</strong></td>
<td>Product Selection</td>
</tr>
</tbody>
</table>

A Selenium “test suite” is a web page with a table of links to web pages with test cases.
Selenium Webdriver

Provides APIs to a variety of languages allowing for very similar capabilities:

```
Select select = new Select(driver.findElement(By.tagName("select")));
select.deselectAll();
select.selectByVisibleText("Edam");
```

Waiting

```
WebDriver driver = new FirefoxDriver();
driver.get("http://somedomain/url_that_delays_loading");
```
System Testing

```java
WebElement myDynamicElement = (new WebDriverWait(driver, 10))
    .until(ExpectedConditions.presenceOfElementLocated(By.id("myDynamicElement")));
```

Waits up to 10 seconds for an expected element to load