

**CS 535 Object-Oriented Programming  
Fall Semester, 2002  
Doc 11 Testing  
Contents**

Testing .....	3
Johnson's Law .....	3
Why Unit Testing .....	4
When to Write Unit Tests .....	8
SUnit & JUnit.....	9
How to Use SUnit.....	10
TestCase methods of interest.....	13
Using JUnit.....	14
Running JUnit .....	16
assert Methods .....	20
Testing the Tests .....	22
Test Fixtures .....	25
Suites – Multiple Test Classes.....	26
How to Test Exceptions .....	29
Testing and Hidden Methods/State.....	30
How to Test Hidden Methods Directly? .....	32
Method 1: Relax the protection level.....	32
Method 2: Use inner classes.....	33
Method 3: Use reflection.....	34
What to Test.....	35
Testing Network Code .....	37

## References

JUnit Cookbook <http://junit.sourceforge.net/doc/cookbook/cookbook.htm>

JUnit Test Infected: Programmers Love Writing Tests  
<http://junit.sourceforge.net/doc/testinfected/testing.htm>

JUnit Javadoc: <http://www.junit.org/junit/javadoc/3.8/index.htm>

Brian Marick's Testing Web Site: <http://www.testing.com/>

Testing for Programmers, Brian Marick, Available at: <http://www.testing.com/writings.html>

**Copyright** ©, All rights reserved. 2002 SDSU & Roger Whitney, 5500 Campanile Drive, San Diego, CA 92182-7700 USA. OpenContent (<http://www.opencontent.org/opl.shtml>) license defines the copyright on this document.

## Reading Assignment

Test Infected - Programmers Love Writing Tests,  
<http://junit.sourceforge.net/doc/testinfected/testing.htm>

JUnit FAQ, <http://junit.sourceforge.net/doc/faq/faq.htm>

Programming with Assertions,  
<http://java.sun.com/j2se/1.4/docs/guide/lang/assert.html>

## Lecture Source Code

Java – CVS module testExamples

Smalltalk – Store package testExamples

## **Testing**

### **Johnson's Law**

If it is not tested it does not work

### **Types of tests**

- **Unit Tests**

Tests individual code segments

- **Functional Tests**

Test functionality of an application

## Why Unit Testing

If it is not tested it does not work

The more time between coding and testing

- More effort is needed to write tests
- More effort is needed to find bugs
- Fewer bugs are found
- Time is wasted working with buggy code
- Development time increases
- Quality decreases

Without unit tests

- Code integration is a nightmare
- Changing code is a nightmare

## **Why Automated Tests?**

What wrong with:

- Using print statements
- Writing driver program in main
- Writing small sample programs to run code
- Running program and testing it by using it

## **Repeatability & Scalability**

Need testing methods that:

- Work with N programmers working for K months (years)
- Help when modify code 6 months after it was written
- Check impact of code changes in rest of system

Practices that work in a school project may not be usable in industry

Standard industry practices may seem overkill in a school project

Work on building good habits and skills

## **We have a QA Team, so why should I write tests?**

How long does it take QA to test your code?

How much time does your team spend working around bugs before QA tests?

How easy is it to find & correct the errors after QA finds them?

Most programmers have an informal testing process

With a little more work you can develop a useful test suite

## **When to Write Unit Tests**

First write the tests

Then write the code to be tested

Writing tests first saves time

- Makes you clear of the interface & functionality of the code
- Removes temptation to skip tests



## SUnit & JUnit

Free frameworks for Unit testing

SUnit originally written by Kent Beck 1994

- Built into VisualWorks 7.0

JUnit written by Kent Beck & Erich Gamma

- Available at: <http://www.junit.org/>

Already installed in JDK 1.2 on rohan and moria

Ports are available in

.NET	Ada	AppleScript	C
C#	C++	Curl	Delphi
Eiffel	Eiffel	Flash	Forte 4GL
Gemstone/S	Haskell	HTML	Jade
LISP	Objective-C	Oracle	Palm
Perl	Php	PowerBuilder	Python
Ruby	Scheme	Smalltalk	Visual Basic
XML	XSLT		

See <http://www.xprogramming.com/software.htm> to download ports

## How to Use SUnit

1. Make test class a subclass of TestCase
2. Make test methods

The framework treats methods starting with 'test' as test methods

3. Run the tests

You can run the test using TestRunner

TestRunner open

Use Browser SUnit Extensions

Load parcel/RBSUnitExtensions.pcl to run the tests from the browser

# Brower SUnit Extensions

The screenshot shows the SUnit browser interface. At the top is a menu bar with options: Browser, Edit, Find, View, Category, Class, Protocol, Method, Tools, Help. Below the menu is a toolbar with various icons for navigation and editing. A search box labeled 'Find:' is on the right. The main area is divided into three panes: 'Category' (showing System-Dependent, System-Name S, System-Override, System-Printing), 'Hierarchy' (showing Counter and TestCounter), and 'Instance' (showing setup, teardown, testing, testDecrease, testIncrease, testZeroDivid). Below these panes are tabs for 'Source', 'Rewrite', and 'Code Critic'. The 'Source' tab is active, displaying the code for the `testIncrease` method: `self deny: counter isNil.`, `counter increase.`, and `self assert: counter count = 1`. At the bottom, a green status bar indicates 'Passed: 1 run, 0 failed, 0 errors'. To the right of the status bar are buttons for 'Profile', 'Debug', and 'Run'. The bottom-most bar shows 'Method: #testIncrease (testing)', 'Parcel: none', and 'Package: TestExample'.

## Sample TestClass

```
Smalltalk.CS535 defineClass: #TestCounter
  superclass: #{XProgramming.SUnit.TestCase}
  indexedType: #none
  private: false
  instanceVariableNames: 'counter '
  classInstanceVariableNames: "
  imports: "
  category: 'Course-Examples'
```

CS535.TestCounter methodsFor: 'testing'

setUp

```
counter := Counter new.
```

tearDown

```
counter := nil.
```

testDecrease

```
counter decrease.
self assert: counter count = -1.
```

testDecreaseWithShould

```
"Just an example to show should: syntax"
counter decrease.
self should: [counter count = -1].
```

testIncrease

```
self deny: counter isNil.
counter increase.
self assert: counter count = 1.
```

testZeroDivide

```
"Just an example to show should:raise: syntax"
self
  should: [1/0]
  raise: ZeroDivide.
```

self

```
shouldnt: [1/2]
raise: ZeroDivide
```

## TestCase methods of interest

Methods to assert conditions:

assert: aBooleanExpression

deny: aBooleanExpression

should: [aBooleanExpression]

should: [aBooleanExpression] raise: AnExceptionClass

shouldnt: [aBooleanExpression]

shouldnt: [aBooleanExpression] raise: AnExceptionClass

signalFailure: aString

setUp

Called before running each test method in the class.

Used to:

- Open files

- Open database connections

- Create objects needed for test methods

tearDown

Called after running each test method in the class.

Used to:

- Close files

- Close database connections

- Nil out references to objects

## Using JUnit Example

Goal: Implement a Stack containing integers.

Tests:

- Subclass junit.framework.TestCase

- Methods starting with 'test' are run by TestRunner

First tests for the constructors:

```
public class TestStack extends TestCase {

    //required constructor
    public TestStack(String name) {
        super(name);
    }

    public void testDefaultConstructor() {
        Stack test = new Stack();
        assertTrue("Default constructor", test.isEmpty() );
    }

    public void testSizeConstructor() {
        Stack test = new Stack(5);
        assertTrue( test.isEmpty() );
    }

}
```

## First part of the Stack

```
package example;
```

```
public class Stack {  
    int[] elements;  
    int topElement = -1;
```

```
    public Stack() {  
        this(10);  
    }
```

```
    public Stack(int size) {  
        elements = new int[size];  
    }
```

```
    public boolean isEmpty() {  
        return topElement == -1;  
    }  
}
```

## Running JUnit

JUnit has three interfaces

- Text (junit.textui.\*)

Fastest to run

- AWT (junit.ui.\*)
- Swing (junit.swingui.\*)

Can reload class files so you can

Run TestRunner once

Recompile program until it passes tests



## Starting Swingui TestRunner

Make sure your classpath includes the code to tested

On Rohan use:

```
java junit.swingui.TestRunner
```

You get a window similar to that on the next page

Enter the full name of the test class

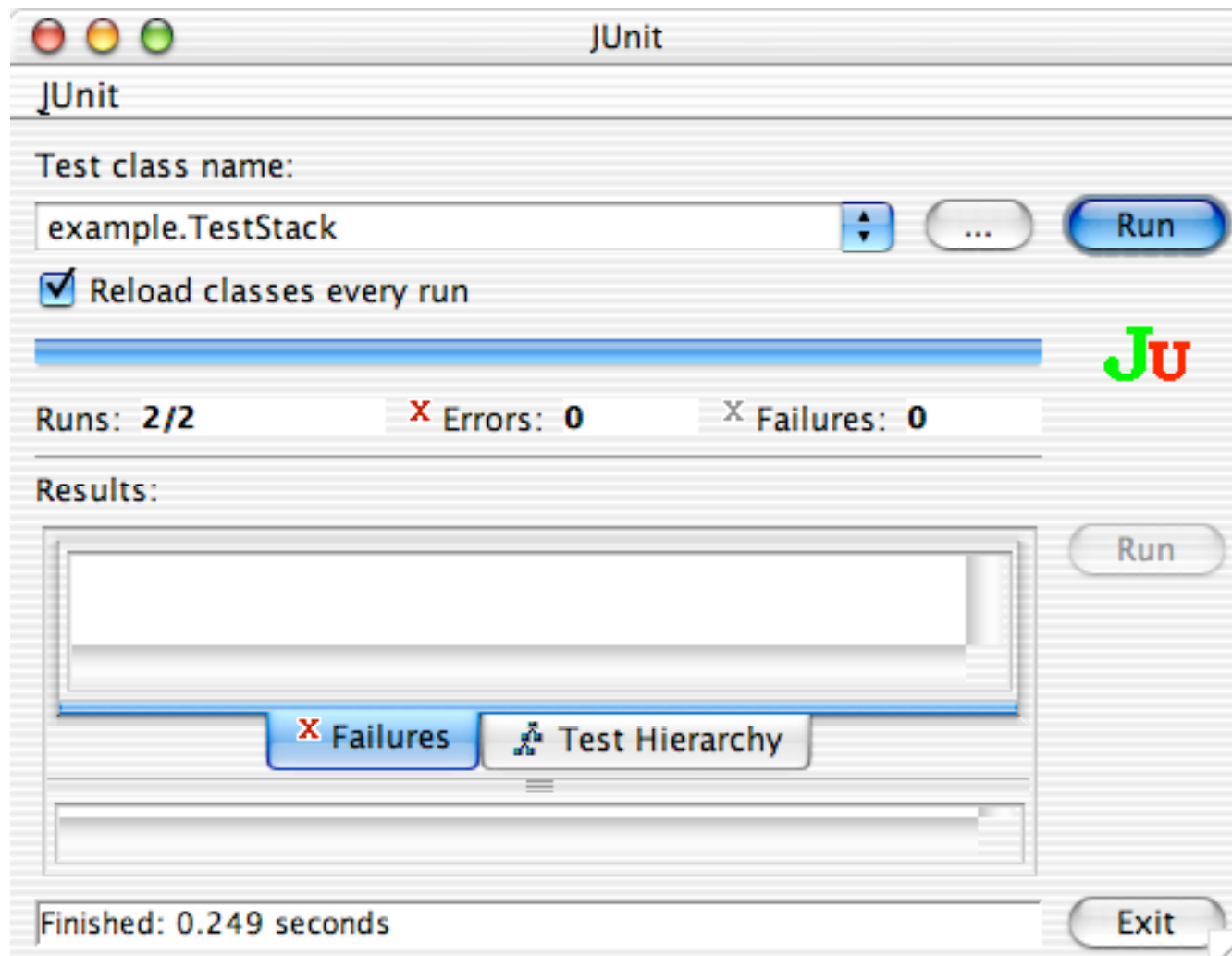
Click on the Run button

If there are errors/failures select one

You will see a stack trace of the error

The “...” button will search for all test classes in your classpath

## Swing version of JUnit TestRunner



## Running the textui TestRunner

Sample Program using main

```
public class Testing {  
    public static void main (String args[]) {  
        junit.textui.TestRunner.run( example.TestStack.class);  
    }  
}
```

### Output

```
..  
Time: 0.067
```

```
OK (2 tests)
```

```
java has exited with status 0.
```

## assert Methods

assertTrue()  
assertFalse()  
assertEquals()  
assertNotEquals()  
assertSame()  
assertNotSame()  
assertNull()  
assertNotNull()  
fail()

For a complete list of the assert methods & arguments see

<http://www.junit.org/junit/javadoc/3.8/index.htm/allclasses-frame.html/junit/junit/framework/Assert.html/Assert.html>

## **JUnit, Java 1.4 & assert**

JUnit had a method called `assert()`

Java 1.4 makes `assert` a reserved word

JUnit starting 3.7 replaces `assert()` with `assertTrue()`

Use JUnit 3.7 or later with JDK 1.4

To use JDK 1.4 asserts:

- Compile with option `-source 1.4`

```
java -source 1.4 programFile.java
```

- Run with the option `-ea`

```
java -ea programFile
```

## Testing the Tests

If can be useful to modify the code to break the tests

package example;

```
public class Stack {  
    int[] elements;  
    int topElement = -1;
```

etc.

```
    public boolean isEmpty() {  
        return topElement == 1;  
    }  
}
```

## Result of running Textui.TestRunner

.F.F

Time: 0.113

There were 2 failures:

1)

testDefaultConstructor(example.TestStack)junit.framework.AssertionFailedError: Default constructor

at example.TestStack.testDefaultConstructor(TestStack.java:22)

at Testing.main(Testing.java:14)

2)

testSizeConstructor(example.TestStack)junit.framework.AssertionFailedError

at example.TestStack.testSizeConstructor(TestStack.java:27)

at Testing.main(Testing.java:14)

FAILURES!!!

Tests run: 2, Failures: 2, Errors: 0

java has exited with status 0.

## **Why Test the Tests?**

One company had an automatic build and test cycle that ran at night. The daily build was created and all the tests were run at night. The test results were available first thing in the morning. One night the build process crashed, so the daily build was not made. Hence there was no code to test. Still 70% of the tests passed. If they had tested their tests, they would have discovered immediately that their tests were broken.



## Test Fixtures

Before each test setUp() is run

After each test tearDown() is run

package example;

```
import junit.framework.TestCase;
```

```
public class StackTest extends TestCase {
```

```
    Stack test;
```

```
    public StackTest(String name) {
```

```
        super(name);
```

```
    }
```

```
    public void setUp() {
```

```
        test = new Stack(5);
```

```
        for (int k = 1; k <=5;k++)
```

```
            test.push( k);
```

```
    }
```

```
    public void testPushPop() {
```

```
        for (int k = 5; k >= 1; k--)
```

```
            assertEquals( "Pop fail on element " + k, test.pop() , k);
```

```
    }
```

```
}
```

## Suites – Multiple Test Classes

Multiple test classes can be run at the same time

Add Queue & TestQueue to Stack classes

```
package example;
```

```
import junit.framework.TestCase;
```

```
public class TestQueue extends TestCase{
    public TestQueue ( String name){
        super(name);
    }

    public void testConstructor() {
        Queue test = new Queue();
        assert( test.isEmpty());
    }
}
```

```
package example;
```

```
import java.util.Vector;
```

```
public class Queue{
    Vector elements = new Vector();
    public boolean isEmpty() {
        return elements.isEmpty();
    }
}
```

## Using a Suite to Run Multiple Test Classes

Running AllTests in TestRunner runs the test in

StackTest  
QueueTest

```
package example;
import junit.framework.TestSuite;
import junit.textui.TestRunner;

public class AllTests {
    static public void main(String[] args) {
        TestRunner.run( example.AllTests.suite());
    }

    static public TestSuite suite() {
        TestSuite suite= new TestSuite();
        Try {
            suite.addTest(new TestSuite(StackTest.class));
            suite.addTest(new TestSuite(QueueTest.class));
        } catch (Exception e){
        }
        return suite;
    }
}
```

## Using Main

We can use main to run the test via `textui.TestRunner`

The command:

```
java example.AllTests
```

will run all the tests in `StackTest` & `QueueTest`

```
package example;
```

```
import junit.framework.TestSuite;
import junit.textui.TestRunner;
```

```
public class AllTests
{
    static public void main(String[] args)
    {
        TestRunner.run(AllTests.class);
    }

    static public TestSuite suite()
    {
        same as last page
    }
}
```

## How to Test Exceptions

At times you may wish to test input to methods that will cause an exception to be thrown

Here is an example of a test that

- passes when an exception is thrown
- fails when the exception is not thrown

Example is from the JUnit FAQ

```
public void testIndexOutOfBoundsException() {  
  
    ArrayList list = new ArrayList(10);  
  
    try {  
  
        Object o = list.get(11);  
  
        fail("Should raise an IndexOutOfBoundsException");  
  
    } catch (IndexOutOfBoundsException success) {}  
}
```

## **Testing and Hidden Methods/State**

### Issues:

- How does one test hidden methods?
- Direct access to an object's state can reduce the time needed to write a test

## Testing Hidden Methods One Position Don't Do it

### Pro:

- Can not test everything
- Clients of an object only care if public interface works correctly
- Testing public interface also tests hidden methods
- Hidden methods are more likely to change, requiring changes to the tests

The basic idea is to work smarter not harder. One cannot completely test each class, and one does not have infinite time to write tests one should write the most effective tests possible. Tests of the public interface of class will also test hidden methods of the class. Bugs in hidden methods that never affect the public methods are not a problem. Since the most important thing is that the public interface works correctly, concentrate your tests on the public interface.

### Con:

- The closer the test is to the code it tests the easier the test
- Bugs in hidden methods can make it hard to debug public methods

My experience is that the more code I write without tests, the more time I spend on finding and correcting bugs. How many times have you spent hours (days?) tracking down a bug that turned out to be a simple bug in some simple untested method, which would have been easy to test? The argument that one cannot test everything and must make effective use of one's testing time is correct. Given the differences in programmer skill level, programmer experience, etc. everyone has to work out their own solution to this. The XP solution is to try to test everything that could possibly break. Since most students are not used to testing, you have to fight the habit of not testing and testing after you have completely finished coding. Given the current state of affairs in commercial software, the industry has a lot to learn about testing.

## How to Test Hidden Methods Directly?

### Method 1: Relax the protection level

In Java one can

- Make the method package level access
- Place the test class in the same package as the tested code

Pro:

- Makes it possible to test the hidden method
- Clients outside the package can not access the method

Con:

- Clients in the package may then use the method
- Requires organizational discipline to avoid using the method

You should comment the method to inform the clients that the method is not to be used



## How to Test Hidden Methods Directly? Method 2: Use inner classes

```
import junit.framework.TestCase;

public class Foo {
    private int value;

    private void bar() {
        value = 10;
    }

    public static class FooTest extends TestCase {
        public FooTest(String name) {
            super(name );
        }

        public void testBar() {
            Foo a = new Foo();
            a.bar();
            assert( 10 == a.value );
        }
    }
}
```

### Pro:

- Provides access to all methods/fields
- Test does not have to be shipped with production code
- Test stays with tested class

### Con:

- Source files are bit harder to read
- Must remove all inner \$class files from production code
- Test not with other test classes

## How to Test Hidden Methods Directly?

### Method 3: Use reflection

#### Pro:

- Java reflection provides access to all methods/fields of a class
- Does not require any changes to tested class

#### Con:

- Reflection can be slow
- Reflection is cumbersome to use
- Requires setting permission files

#### See

<http://www.eli.sdsu.edu/courses/fall98/cs596/notes/reflection/reflection.html> for more information about reflection

## **What to Test**

Everything that could possibly break

Test values

- Inside valid range

- Outside valid range

- On the boundary between valid/invalid

GUIs are very hard to test

- Keep GUI layer very thin

- Unit test program behind the GUI, not the GUI

## Common Things that Programs Handle Incorrectly

Adapted with permission from “A Short Catalog of Test Ideas” by Brian Marick, <http://www.testing.com/writings.html>

### Strings

Test using empty String

### Collections

Test using:

- Empty Collection
- Collection with one element
- Collection with duplicate elements
- Collections with maximum possible size

### Numbers

Test using:

- Zero
- The smallest number
- Just below the smallest number
- The largest number
- Just above the largest number

## **Testing Network Code**

Writing automated tests for network code can be hard

Make the network code very thin

Write automated tests for the non-network code

## Example – SimpleDateServer

The SimpleDateServer has two methods

- One with network code
- One for handling the request with no network code

Write tests for processRequestOn:

```
SimpleDateServer>>run
```

```
| childSocket clientIOStream |
```

```
[childSocket := serverSocket accept.
```

```
clientIOStream := childSocket readAppendStream.
```

```
clientIOStream lineEndTransparent.
```

```
self processRequestOn: clientIOStream.] repeat
```

```
SimpleDateServer>>processRequestOn: anReadAppendStream
```

```
| clientRequest |
```

```
clientRequest := anReadAppendStream through: Character cr.
```

```
(clientRequest startsWith: 'date')
```

```
ifTrue:
```

```
    [anReadAppendStream
```

```
        nextPutAll: Time dateAndTimeNow printString;
```

```
        commit].
```

```
anReadAppendStream close
```

## **Issues in Writing test for processRequestOn:**

### **processRequestOn: Input**

Need to create an ExternalAppendStream for input

Would be easier to test if arguments were:

- ReadStream
- WriteStream

### **Server Output**

Initial sever was written without thought to parsing output

It will be hard for tests to parse current output

So it will be hard for a client to parse server response

Conclusion:

- Change server to make writing tests & client easier

## Modified Server

```
SimpleDateServer>>run
```

```
  | childSocket |
```

```
  [childSocket := serverSocket accept.
```

```
  self
```

```
    processRequest: childSocket readStream lineEndTransparent
```

```
    response: childSocket writeStream lineEndTransparent]
```

```
    repeat
```

```
SimpleDateServer>>processRequest: aReadStream response:
```

```
aWriteStream
```

```
  | clientRequest |
```

```
  clientRequest := aReadStream through: Character cr.
```

```
  (clientRequest startsWith: 'date')
```

```
    ifTrue:
```

```
      [aWriteStream
```

```
        nextPutAll: Date today printString;
```

```
        commit].
```

```
  (clientRequest startsWith: 'time')
```

```
    ifTrue:
```

```
      [aWriteStream
```

```
        nextPutAll: Time now printString;
```

```
        commit].
```

```
  aWriteStream close.
```



## Some Tests

### TestSimpleDateServer is subclass of TestCase

TestSimpleDateServer>>testDate

```
| command response server serverDate serverDateString |
command := 'date \' withCRs readStream.
response := WriteStream on: String new.
server := SimpleDateServer new.
server processRequest: command response: response.
serverDateString := response contents.
serverDate :=Date readFrom: serverDateString readStream.
self assert: serverDate = Date today.
```

TestSimpleDateServer>>testTime

```
| command response server serverTimeString serverTime |
command := 'time \' withCRs readStream.
response := WriteStream on: String new.
server := SimpleDateServer new.
server processRequest: command response: response.
serverTimeString := response contents.
serverTime :=Time readFrom: serverTimeString readStream.
self assert: (Time now asSeconds - serverTime asSeconds) < 2 .
```

TestSimpleDateServer>>testBadInput

```
| command response server |
command := 'fooBar \' withCRs readStream.
response := WriteStream on: String new.
server := SimpleDateServer new.
server processRequest: command response: response.
self assert: response contents trimBlanks isEmpty
```

## Lessons

Keep the network layer thin

Server design may need to be modified to make it testable

See

Test networked code the easy way, **Nelson Minar**

<http://www.javaworld.com/javaworld/jw-07-2002/jw-0719-networkunittest.html>

for another example