

**Software Engineering 2005**

# **Testing Tools and Techniques**

Martin Bravenboer

Department of Information and Computing Sciences

Universiteit Utrecht

`martin@cs.uu.nl`

October 18, 2005

# Testing Automation

---

*“Design your world so that you can’t help but succeed”*

Creating a **no-fail environment**

- Version Control
- Testing
- Automation

**As Seen On**



Testing from *“Coding in Fear”* to *“Coding with Confidence”*

# Testing Tools and Techniques

---

1. Unit Testing with JUnit
2. Mock Techniques
3. Domain Specific Unit Testing Tools
4. Code Coverage Analyzers
5. Continuous Integration Tools
6. Design for Testing

## Categories of Tools (1)

---

- General purpose test frameworks
  - JUnit
  - NUnit
- Support libraries
  - Mock Objects
- Domain-specific test frameworks
  - HttpUnit
  - XmlUnit
  - JfcUnit
- Memory management analyzers
  - Valgrind

## Categories of Tools (2)

---

- Code coverage analysers
  - Clover
  - Emma
- Test generators
  - jTest
- Stress Testing Tools
  - LoadSim, JUnitPerf
- Static analysis
  - Lint
  - FindBugs
  - Checkstyle

# **JUnit**

# xUnit

---

- **xUnit** Family
  - Simplicity
  - Define, run, and check
- Instance: **JUnit**
  - Very simple: 5 classes in `junit.framework`
  - Many extensions
  - <http://www.junit.org>
- Instance: **NUnit**
  - .NET Attributes
  - <http://nunit.sf.net>

# JUnit: Example

---

```
import junit.framework.TestCase;

public class TestSimple extends TestCase {

    public TestSimple(String name) {
        super(name);
    }

    public void testSplitEmpty() {
        String[] result = " ".split("\\s+");
        assertEquals("Should not produce a token", 0, result.length);
    }

    public void testSplit2() {
        String[] result = "a  bc ".split("\\s+");
        assertEquals("Should be 2 tokens", 2, result.length);
    }
}
```

# JUnit: Grouping Tests

---

## TestSuite

- `addTest(Test)`
- `addTestSuite(Class)`

How to construct a Test?

```
new MyTest("testFoo")
```

or

```
new MyTest("Some text") {  
    public void runTest() {  
        testFoo();  
    }  
}
```

# JUnit: Grouping Tests

---

```
public class TestFoo extends TestCase {  
  
    public TestFoo(String method) { ... }  
  
    public void testFoo() { ... }  
    public void testBar() { ... }  
  
    public static Test suite() {  
        TestSuite result = new TestSuite();  
        result.addTest(new TestFoo("testFoo"));  
        result.addTest(new TestFoo("testBar"));  
        return result;  
    }  
}
```

or:

```
return new TestSuite(TestFoo.class);
```

## JUnit: Grouping Tests

---

Typical composition class:

```
public class TestAll {
    public static Test suite() {
        TestSuite result = new TestSuite("All tests");
        result.addTestSuite(TestFoo.class);
        result.addTestSuite(TestBar.class);
        return result;
    }
}
```

# JUnit: Running a Test

---

## TestRunner

- invokes `suite` method
- or constructs `TestSuite`

## Graphical

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

## Textual

```
$ java junit.textui.TestRunner
```

## JUnit: Assertions

---

*“Real testing checks results”*

- `assertTrue`, `assertFalse`
- `assertEquals`
- `assertSame`
- `assertNotNull`
- `fail`

Advice: Use `assert*` with a failure message.

## JUnit: Fixture per Test

---

```
private XMLReader _reader;

protected void setUp() throws Exception {
    SAXParserFactory factory = SAXParserFactory.newInstance();
    factory.setNamespaceAware(true);
    factory.setValidating(false);
    _reader = factory.newSAXParser().getXMLReader();
}

protected void tearDown() {
    _reader = null;
}

public void testFoo () throws Exception {
    _reader.parse(new InputSource(new StringReader("<Foo/>")));
    // I want to assert something :(
}
```

Fixture `setUp` and `tearDown` is performed for every test.

## JUnit: Fixture per Suite

---

```
TestSetup wrapper = new TestSetup(suite) {
    protected void setUp() {
        oneTimeSetUp();
    }

    protected void tearDown() {
        oneTimeTearDown();
    }
};
```

Don't use static initializers.

Must be in static variables.

## JUnit: (Expected) Exceptions

---

Not expected: don't catch, just throw!

```
public void testRead () throws IOException {
    StringReader reader = new StringReader("SWE!");
    assertEquals("First char must be S", reader.read(), 'S');
}
```

Expected: try-fail-catch

```
public void testList () {
    List<String> list = new ArrayList<String>();
    list.add("Foo");

    try {
        String result = list.get(1);
        fail("Not a valid index.");
    } catch (IndexOutOfBoundsException exc) {
        assertTrue(true);
    }
}
```

## JUnit: Best Practices (1)

---

- Make a test fail now and then  
*“Spring the Trap”*
- Keep tests **independent**
  - No order guarantee
  - Separate instances for every test
- Custom, domain-specific **asserts**
- **Design** test code.
  - Production/Test : 50/50
  - Tests are not a list of statements.

## JUnit: Best Practices (2)

---

- Test just **one** object
  - Mock up related objects
- Write test back (asserts) first.
- Don't use the same value twice (1 + 1)
- Don't break **encapsulation** for testing.
  - Test code in same package
  - Use parallel source trees

## JUnit: Best Practices (3)

---

- Do you need to search for a bug? Wrong!
- Write tests that **reduce the amount of code** where the problem might be.
- Test just one thing: Faulty code  $\Leftrightarrow$  Test
- Never **ever** modify an existing to test a new situation
- Testing is not an end in itself

## JUnit: More Automation

---

Why not build in IDE?

- Poor automation, building from source, deployment

⇒ Integrate in automated build process.

```
<junit printsummary="yes" haltonfailure="yes">
  <classpath>
    <path refid="project.cp"/>
    <pathelement location="{build.tests}"/>
  </classpath>
  <formatter type="plain"/>
  <test name="org.foo.TestAll" />
</junit>
```

**batchtest**: compose tests in Ant

# JUnit: Fancy Reports

---

## Formatter

```
<junit ...>
  ...
  <formatter type="xml"/>
  <test todir="$builddir/reports" ... >
</junit>
```

## Report Generation

```
<junitreport>
  <fileset dir="$builddir/reports">
    <include name="TEST-*.xml"/>
  </fileset>

  <report todir="$builddir/reports/html"/>
</junitreport>
```

## JUnit: Further Reading

---

- *“Pragmatic Unit Testing in Java and JUnit”*
- *“JUnit in Action”*
- *“Java Open Source Programming with XDoclet, JUnit, WebWork, Hibernate”*
- `http://www.junit.org`

# Mock Techniques

## Mock Objects: Rationale

---

- Test just one thing
- Tests should focus. Be Independent.
- Faulty code  $\Leftrightarrow$  Test

But ... What if code is not just 'one thing'?

$\Rightarrow$  **Fake** the other 'things'!

$\Rightarrow$  **Mock objects**

## Use Mock Objects If ...

---

- Behaviour that is beyond your control.
- Exceptional situations (difficult to reproduce)
- External resources (server, database)
- Lot of code to setup.
- Expensive, poor performance
- Reflection and interaction: How was code used?
- Non-existing code.

## Solution: Test the Interactions

---

- More difficult with state
  - Is the content type set?
  - Is the 'temp directory' method not invoked twice in a session?
  - Is a topic scheduled for indexing?
  - Is a stream closed?
- Almost impossible with state
  - Is the request's pathinfo requested when ShowFile is invoked?
  - Are the access flags checked when a topic is requested?
  - Is the log function invoked with an error?

⇒ Communicate with testcase instead of 'real code'

# Mock Object Generators

---

## At compile-time

- MockMaker – <http://mockmaker.sf.net>
- MockCreator – <http://mockcreator.sf.net>

## At runtime

- DynaMock – <http://www.mockobjects.com>
- jMock – <http://www.jmock.org>
- EasyMock – <http://www.easymock.org>

## Features

- Define Expectations
- Mock-ready implementations of standard libraries

# Mock Objects: EasyMock

---

Expectations as actual method calls!

Mock Control

- Record and replay mode.
- Control, StrictControl, NiceControl

MockControl methods

- `getMock()`
- `replay()`, `verify()`
- `setReturnValue`
- `setThrowable(...)` (**Crash Test Dummy**)

<http://www.easymock.org>

# EasyMock: Example 1

---

```
private XMLReader _reader;
private MockControl _control;
private ContentHandler _handler;

protected void setUp() throws Exception {
    ...

    _control = MockControl.createNiceControl(ContentHandler.class);
    _handler = (ContentHandler) _control.getMock();
    _reader.setContentHandler(_handler);
}

public void testFoo () throws Exception {
    _handler.endElement("", "Foo", "Foo");
    _control.replay();

    _reader.parse(new InputSource(new StringReader("<Foo/>")));
    _control.verify();
}
```

## EasyMock: Example 2

---

```
protected void setUp() throws Exception {
    ...
    _control = MockControl.createStrictControl(ErrorHandler.class);
    _handler = (ErrorHandler) _control.getMock();
    _reader.setErrorHandler(_handler);
}

public void testIllegalFoo() throws Exception {
    _handler.fatalError(null);
    _control.setMatcher(MockControl.ALWAYS_MATCHER);
    _control.replay();

    try {
        _reader.parse(new InputSource(new StringReader("<foo></bar>")));
    } catch(SAXParseException exc) {
        assertTrue(true); <---
    }
    _control.verify();
}
```

## Mock Objects: Limitations

---

- Requires an interface: implementation must be substitutable.
  - Design for Testing
  - Example: test if a `strong` element is created.  
Use `JDOMFactory`
- Requires 'mock-ready' libraries. Many libraries do not allow this kind of configuration.
- DynaMock: no type checking by compiler.
- DynaMock: verbose constraints and return values.
- Easy to break the real code: what does it use?

## Mock Objects: Further Reading

---

- *“Endo Testing: Unit Testing with Mock Objects”* (XP2000)
- *“Mock Objects”* in *“Pragmatic Unit Testing”*  
Sample chapter available online
- [www.mockobjects.com](http://www.mockobjects.com)

# Domain Specific Tools

## Some Popular JUnit Extensions

---

- Web Applications  
e.g. HttpUnit
- Performance  
e.g. JUnitPerf
- Integration  
e.g. Cactus (J2EE)
- GUI Applications  
e.g. JfcUnit
- XML Processing  
e.g. XML Unit

# JUnitPerf: Test Decorators

---

## TimedTest

```
Test testCase = ... ;  
Test timedTest = new TimedTest(testCase, 2000);
```

## LoadTest

```
Test testCase = ... ;  
Test loadTest = new LoadTest(testCase, 25);
```

## RepeatedTest

```
Test testCase = ...;  
Test repeatedTest = new RepeatedTest(testCase, 10);  
Timer timer = new ConstantTimer(1000)  
Test loadTest = new LoadTest(repeatedTest, 25, timer);
```

<http://www.clarkware.com/software/JUnitPerf.html>

# XML Processing: XMLUnit

---

Handy utils for XML, XPath, XSLT.

XMLTestCase: `assertXMLEqual`

- Identical / similar, meaningful diff messages

Transform

- Easy to use in XSLT tests

Validator

- Tools for DTD and Schema validation

<http://xmlunit.sf.net>

No longer maintained :(

# Parsing: ParseUnit

---

```
testsuite Expressions
```

```
topsort Exp
```

```
test simple addition
```

```
"2 + 3" -> Plus(Int("2"), Int("3"))
```

```
test addition is left associative
```

```
"1 + 2 + 3" -> Plus(Plus(Int("1"), Int("2")), Int("3"))
```

```
test multiplication has higher priority than addition
```

```
"1 + 2 * 3" -> Plus(Int("1"), Mul(Int("2"), Int("3")))
```

```
test
```

```
"x1" succeeds
```

```
test
```

```
"1x" fails
```

# Testing Web Applications

## Some Issues

---

- Security
  - Resources protected?
  - Unexpected input?
  - ... not generated by a browser?
- Validation of XHTML, CSS, JavaScript
- Check broken links
- Are files served correctly? (type)
- Does my caching mechanism work?
- Can you handle loads of users?

# Unit Testing of Java Web Applications

---

- **Servlet Unit Testing**
  - Mock objects for servlets container
  - Servlet container: difficult to mock
  - Mock Objects Framework implements servlet mocks
- **In-Container Integration Testing**
  - Cactus (J2EE integration tests)
  - Rather heavy-weight
- **Functional Testing**
  - Simulate browser
  - HttpUnit

# Servlet Testing with Mock Objects

---

## Approach

- Mock the objects on which a servlet depends.
- Mock the servlet container.

## Implementation (mockobjects.com)

- `MockHttpServletRequest`
  - Supports expectations  
e.g. `request.setExpectedError(SC_NOT_FOUND)`
  - Verify: `request.verify()`;
- `MockServletConfig`
- `MockServletInputStream`
- ...

## **Servlet Container Mock: (Dis)advantages**

---

- Possible to mock the dependencies of servlets
  - mock database
  - mock logger to check if 404 was logged
- Tends to be close to functional testing
- Why not test at the client-side?
  - Difficult to mock dependencies of servlet

In general: Leave servlet specific environment as soon as possible.

# HttpUnit: Functional Testing

---

- Automated functional tests
  - ⇒ No need for checking the site by hand.
- Targets programmers
- Simplicity. Just code.
- Well-designed web applications:
  - Request/response is unit
  - Possible to test

Supports frames, forms, JavaScript, basic HTTP authentication, cookies and redirection.

`http://httpunit.sf.net`

# HttpUnit: Hello World

---

- WebClient / WebConversation
- WebRequest
- WebResponse

```
WebClient client = new WebConversation();
WebResponse response = client.getResponse("http://www.cs.uu.nl/wiki/Gw");
WebLink link = response.getLinkWith("Vision");
WebRequest request = link.getRequest();
```

## Links:

- WebLink[] getLinks
- WebLink getLinkWith
- WebLink getLinkWithID
- WebLink getLinkWithImageText

## HttpUnit: Working with Forms

---

Retrieving values from a form:

```
WebForm form = response.getForms()[0];
assertEquals("Martin Bravenboer", form.getParameterValue("name"));
assertEquals("", form.getParameterValue("password"));
assertEquals("", form.getParameterValue("password2"));
```

Submitting a form:

```
form.setParameter("name", "Martin Bravenboer");
form.setParameter("password", "foobar");
form.setParameter("password2", "foobar");
form.submit();
```

## Testing Web Applications: Further Reading

---

- <http://www.c2.com/cgi/wiki?ServletTesting>
- *“Testing a Servlet”* in *“Mock Objects”* in *“Pragmatic Unit Testing”*
- *“Unit Testing Servlets and Filters”* in *“JUnit in Action”*
- *“In-container Testing with Cactus”* in *“JUnit in Action”*
- *“Functional Testing with HttpUnit”* in *“Java Tools for Extreme Programming”*

# Code Coverage Analyzers

# Code Coverage Analysis

---

- How well is the code exercised?
- Improving Coverage
  - Write more tests
  - Refactoring: Eliminate duplication, more reuse

## Implementations

- Clover – Closed Source. Free for non commercial use  
<http://www.cenqua.com/clover/index.html>
- JCoverage – GPL/Closed Source <http://www.jcoverage.com>
- Quilt – Open Source <http://quilt.sf.net>
- Emma – Open Source <http://emma.sf.net>
- Coverlipse – Open Source <http://coverlipse.sf.net>

## Clover coverage report - Checkstyle Project

Coverage timestamp: Thu Aug 26 2004 12:35:30 EST

Overview [Package](#) File

[FRAMES](#) [NO FRAMES](#)

package stats: LOC: 4,649 Methods: 213

NCLOC: 2,974 Classes: 34

Files: 26

Package	Conditionals	Statements	Methods	TOTAL
<b>com.puppycrawl.tools.checkstyle</b>	67.6%	79.7%	87.8%	<b>78.8%</b>

Classes	Conditionals	Statements	Methods	TOTAL
<a href="#">AllTests</a>	-	0%	0%	<b>0%</b>
<a href="#">Main</a>	0%	0%	0%	<b>0%</b>
<a href="#">PropertyCacheFile</a>	21.4%	14%	66.7%	<b>19.5%</b>
<a href="#">TreeWalker.SilentJava14Recognizer</a>	-	100%	50%	<b>60%</b>
<a href="#">DefaultLogger</a>	58.3%	80.6%	55.6%	<b>71.2%</b>
<a href="#">PropertiesExpander</a>	50%	75%	100%	<b>75%</b>
<a href="#">Checker</a>	63.2%	83.4%	87.5%	<b>78.1%</b>
<a href="#">Checker.ErrorCounter</a>	50%	80%	85.7%	<b>78.6%</b>
<a href="#">TreeWalker</a>	87.5%	80.3%	90%	<b>82.8%</b>
<a href="#">DebugAuditAdapter</a>	-	87.5%	87.5%	<b>87.5%</b>
<a href="#">PackageNamesLoader</a>	80%	88.6%	100%	<b>88.7%</b>
<a href="#">DefaultConfiguration</a>	50%	90%	100%	<b>89.5%</b>
<a href="#">PackageNamesLoaderTest</a>	-	86.7%	100%	<b>89.5%</b>
<a href="#">ConfigurationLoader</a>	87.5%	91.3%	100%	<b>91%</b>
<a href="#">ConfigurationLoaderTest</a>	100%	91.8%	100%	<b>93.5%</b>
<a href="#">ConfigurationLoader.InternalLoader</a>	90%	95%	100%	<b>93.9%</b>
<a href="#">PackageObjectFactoryTest</a>	-	91.7%	100%	<b>94.1%</b>
<a href="#">UtilsTest</a>	-	94.4%	100%	<b>95%</b>
<a href="#">XMLLogger</a>	90%	98.7%	100%	<b>97.1%</b>
<a href="#">StringArrayReaderTest</a>	-	98.1%	100%	<b>98.2%</b>
<a href="#">CheckerTest</a>	50%	100%	100%	<b>99.2%</b>
<a href="#">BaseCheckTestCase</a>	100%	100%	100%	<b>100%</b>
<a href="#">BaseCheckTestCase.BriefLogger</a>	-	100%	100%	<b>100%</b>
<a href="#">DebugChecker</a>	-	100%	100%	<b>100%</b>
<a href="#">DebugFilter</a>	-	100%	100%	<b>100%</b>
<a href="#">DefaultContext</a>	-	100%	100%	<b>100%</b>
<a href="#">OptionTest</a>	-	100%	100%	<b>100%</b>
<a href="#">PackageObjectFactory</a>	100%	100%	100%	<b>100%</b>
<a href="#">StringArrayReader</a>	100%	100%	100%	<b>100%</b>
<a href="#">XMLLoggerTest</a>	100%	100%	100%	<b>100%</b>
<a href="#">XMLLoggerTest.TestThrowable</a>	-	100%	100%	<b>100%</b>

## Clover: Example

---

```
120 0      public void addException(AuditEvent aEvt, Throwable aThrowable)
121      {
122 0      synchronized (mErrorWriter) {
123 0          mErrorWriter.println("Error auditing " + aEvt.getFileName());
124 0          aThrowable.printStackTrace(mErrorWriter);
125      }
126  }
127

154 347    protected void closeStreams()
155      {
156 347        mInfoWriter.flush();
157 347        if (mCloseInfo) {
158 347            mInfoWriter.close();
159        }
160
161 347        mErrorWriter.flush();
162 347        if (mCloseError) {
163 0          mErrorWriter.close();
164        }
165    }
```

Java - Computation.java - Eclipse Platform

File Edit Source Refactor Navigate Search Project Run Window Help

Coverlip... »3

Block Coverage  
Clear results

simpletests  
Computation

```

public class Computation {
    public int add(int arg1, int arg2) {
        int result = arg1 + arg2; int meinInt = 0;
        int result2 = result;
        if (arg1 == Integer.MIN_VALUE) {
            new Integer(result);
        }
        int result3 = result2;
        return result + meinInt;
    }

    public int multiply(int n, int m) {
        int result = 0;
        for (int j = 0; j < m; j++) {

```

Problems Javadoc Declaration Console Coverlipse Markers View Synchronize

Show definitions

Message	Line	covered uses	uncovered uses	Resource
This line was fully covered	13			Computation.java
This line was fully covered	14			Computation.java
This line was fully covered	15			Computation.java
This line was not covered	16			Computation.java
This line was fully covered	18			Computation.java
This line was fully covered	19			Computation.java
This line was fully covered	23			Computation.java

# Continuous Integration Tools

# Continuous Integration Tools

---

- Types of Automation
  - Commanded Automation
  - Scheduled Automation
  - Triggered Automation
- Integrate components as early and often as possible.
- Server monitors repository
- Avoid big bang integration
- Requires
  - Automated build process
  - Automated distribution process
  - Automated deploy process

## Continuous Integration: Implementations

---

- CruiseControl (for Java)
- Maven Continuum
- AntHill
- Daily Build System
- Nix Buildfarm
- ...

# Continuous Integration

---

- **AntHill**

- Handle multiple projects and their dependencies
- Time-based build scheduler (instead of commit push)
- <http://www.urbancode.com/projects/anthill/>

- **Daily Build System**

- Shell-based builders and checkout
- Time-based scheduler (by name daily)
- <http://www.program-transformation.org/Tools/DailyBuildSystem>

- **Maven Continuum**

- New project, currently time-based scheduling

⇒ Bad experience with time-based scheduling.

# Continuous Integration: CruiseControl

---

- Repeated builds
- Build jobs implemented in Ant
- Trigger: modification checks  
CVS, Subversion, VSS, file system, HTTP and more.
- Publishers  
Status, Email, SCP, FTP

## Limitations

- SCM logic in build files
- No managing of dependencies (Maven is supported)

<http://cruisecontrol.sf.net>

# Nix Buildfarm

---

- Based on Nix:
  - Full description of dependencies and their configuration
  - Functional abstraction: variants
  - Caching for free
- Builders: arbitrary: shell, ...
- Distributed builds
- Building RPMs in User-Mode Linux
- Triggers: Subversion, HTTP/FTP only
- Builders: abstractions targeted at GNU packages.
- Lack of status interface (student working on that)

<http://nix.cs.uu.nl>

# Nix Buildfarm: Example

---

```
rec {
  trunkRelease    = release "gw-server-trunk"    inputs.gwTrunk;
  storageRelease  = release "gw-server-storage"  inputs.gwStorage;
  blogRelease     = release "gw-server-blog"     inputs.gwBlog;

  release = name : input :
    let {
      body =
        makeReleasePage {
          fullName = "Generalized Wiki Server";
          sourceTarball = myTarBuild;
          nixBuilds = [ server.body ];
          nodistBuilds = [ server.gw ];
          ...
        }
      ...
    };
}
```

# Nix Buildfarm: Example

---

```
rec {
  headRelease = release "bibtex-tools"
                std.head inputs.bibtexToolsHead;

  branch014Release = release "bibtex-tools for Stratego/XT 0.14"
                             std.release_014 inputs.bibtexToolsBranch014;

  release =
    fullName : for : input :
      for.release {
        inherit input distBaseUrl fullName;
        moreBuildInputs = base : [base.pkgs.hevea base.pkgs.tetex];
        moreInputs = base : {
          withhevea = base.pkgs.hevea;
          withlatex = base.pkgs.tetex;
        };
      };
};
}
```

## Continuous Integration: Further Reading

---

- *“Pragmatic Project Automation: How to Build, Deploy, and Monitor Java Applications”*

Excerpts available

- *“Continuous Integration”* (Martin Fowler)

<http://www.martinfowler.com/articles/continuousIntegration.html>

# Design for Testing

# Design for Testing

---

*“Your code sucks if it isn’t testable”*

**Dave Astels, an Artima Blogger**

- Separation of concerns
  - ⇒ Separate, independent tests
  - ⇒ Faulty code easier to identify
- Design for flexibility.
  - Testing requires different implementations
- Unit testing improves the design of code
  - Small methods
  - Reduction of side effects
  - More explicit arguments

## Design for Testing: 'Inversion of Control'

---

- aka 'Dependency Injection', 'Tell, Don't Ask'
- `new SomeComponentB(...)` inside `ComponentA` is bad
- Connecting components
- Don't let objects collect their own stuff
- Push versus pull
- Container/Context: the container sets everything an object needs
- Allows replacement with a Mock.

<http://www.martinfowler.com/articles/injection.html>

## Design for Testing: 'Reduce Coupling'

---

- No static singletons
  - Cannot be replaced
  - Might not be cleared while running several tests
- Use factories instead of explicit instantiation
  - Example: `JDOMFactory`
  - Scope: process, thread, session
- Law of Demeter
  - Method target restricted to 'local' or 'global' objects
  - Makes it easier to mock dependencies

## Design for Testing: 'Gateway'

---

*"Encapsulate access to external systems and resources"*

- Libraries
- Services
- Databases

### Implementation

- Create an API for your usage
- Translate into external resource invocation

### Use *"Separated Interface"*

- Compare to Law of Demeter
- *"Service Stub"* implementation (Mock)

## Design for Testing: Further Reading

---

- *“Patterns of Enterprise Application Architecture”*  
<http://c2.com/cgi/wiki?PatternsOfEnterpriseApplicationArchitecture>
- *“Tell, Don’t Ask”*  
[http://www.pragmaticprogrammer.com/pp11c/papers/1998\\_05.html](http://www.pragmaticprogrammer.com/pp11c/papers/1998_05.html)  
<http://c2.com/cgi/wiki?TellDontAsk>
- *“Law of Demeter”*  
<http://c2.com/cgi/wiki?LawOfDemeter>
- *“Design Patterns: Elements of Reusable Object-Oriented Software”*  
<http://c2.com/cgi/wiki?DesignPatternsBook>