Struts Testing - EXPOSED!

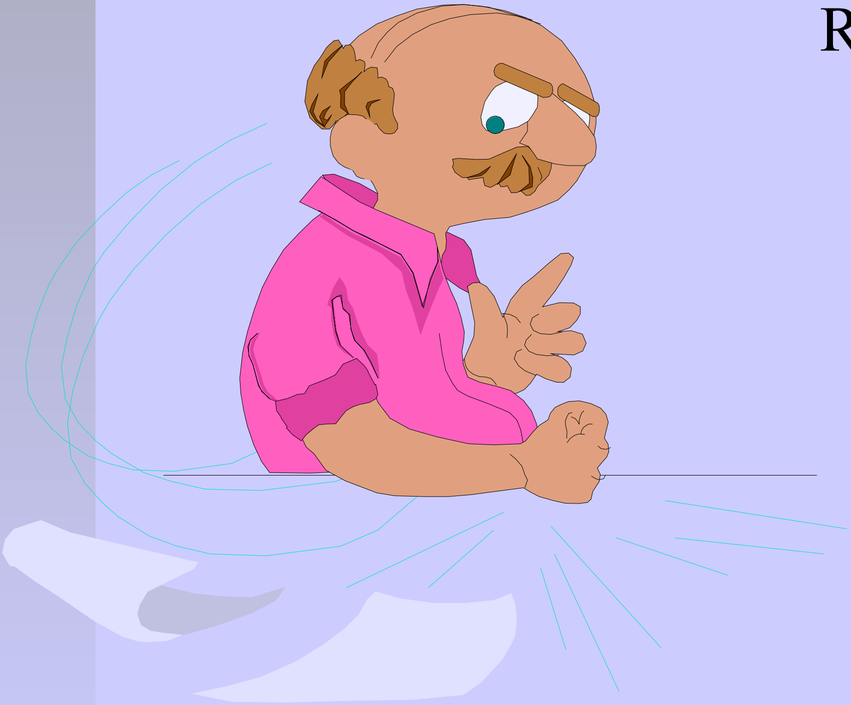
Unit Testing Struts Applications

"Never in the field of software development was so much owed by so many to so few lines of code" -- Martin Fowler on JUnit

Extra! Extra! Read All About It ...

=== Dateline: October 2003 – Atlanta, GA ===

ROGUE DEVELOPERS CAUGHT TESTING CODE

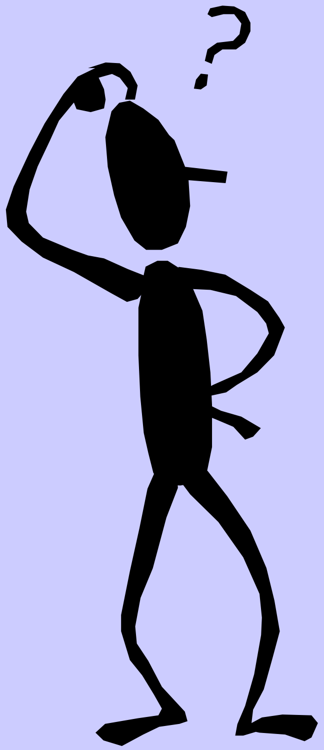


Rogue developers appear to be actually running tests against their code *themselves* – even before the code has been released! Aghast managers fear a massive decline in productivity. Rumors abound of this ***extreme*** behavior. Apparently some insane hackers have been found to be writing tests before they even write the code!

Mission Impossible?

Your mission – should you choose to accept it ...

Investigate this heinous phenomena and report back to HQ ... *ASAP!*
P.S. Don't forget the 5 W's!



- **Who** is doing this testing?
- **What** is being tested? What's a Unit Test?
- **When** is testing occuring?
- **Where** is the testing taking place?
- **Why** test to begin with?

Who Dunnit (doinit)?

- Pioneered by Kent Beck
  - Simple Smalltalk Testing (*Oct. 1994)*
  - Inventor of the xUnit Family of Tools
  - Adopted as an Extreme Programming Core Practice
- Extreme (Agile) Programmers
- Test-Driven Developers (TDD)
- Developers that hate maintaining old code
- RUPers (just don't tell the PMs!)
- YOU! (If not ... You should be!)

What's a Unit Test?

- Code that executes and evaluates the behavior of some software *unit*
- A *Unit* could be a Class, a JSP, a Servlet, an EJB, an HTML page, a Struts Action,...
- A Unit test evaluates *assertions*
  - *Assert – to state or declare positively*
  - *Assertions* are used to verify that the Unit behaves as expected under all conditions
- Unit tests do *not* test load or stress
- A unit test is a *fixture* – i.e. a device that supports work (the unit) during testing
- True unit test *isolates* the subject (i.e. A unit test is not an integration test)

A Simple JUnit Example

The Unit

/app/src/bank/Account.java

```
package bank;

public class Account {

  public int getBalance(){
    return balance;
  }


  void deposit(int amt){
    balance += amt;
  }

  private int balance;
}
```

The Unit Test

/app/test/src/bank/AccountTest.java

```
package bank;
import junit.framework.TestCase;
public class AccountTest
           extends TestCase {

  // executed before each test method
  public void setUp() {
    acct = new Account();
  }
  public void testDeposit() {
    int bal = acct.getBalance();
    int amt = 75; // deposit amount
    acct.deposit(amt);
    assertEquals( bal + amt,
                  acct.getBalance());
  }
  // executed after each test method
  public void tearDown() {}
  private Account acct;
}
```

JUnit/Eclipse Integration

- New JUnit Test Case Wizard
- Provides a Test Runner

JUnit/Ant Integration

- Running JUnit tests
- Generating a JUnit Report

When should you unit test?

- Write the unit test *before* you write the unit!
  - It helps drive the proper API because the unit test is a client to the API (Test-Driven Development)
- Update the Unit Test whenever you change the unit.
- Run the test when you change the unit.
  - IDE Integration/plug-ins help here
- Run the unit test when you build.
- Run the unit test when *externals* change.
  - Upgrading/replacing third-party software
  - Changing the database schema
  - Deploying to a different application server

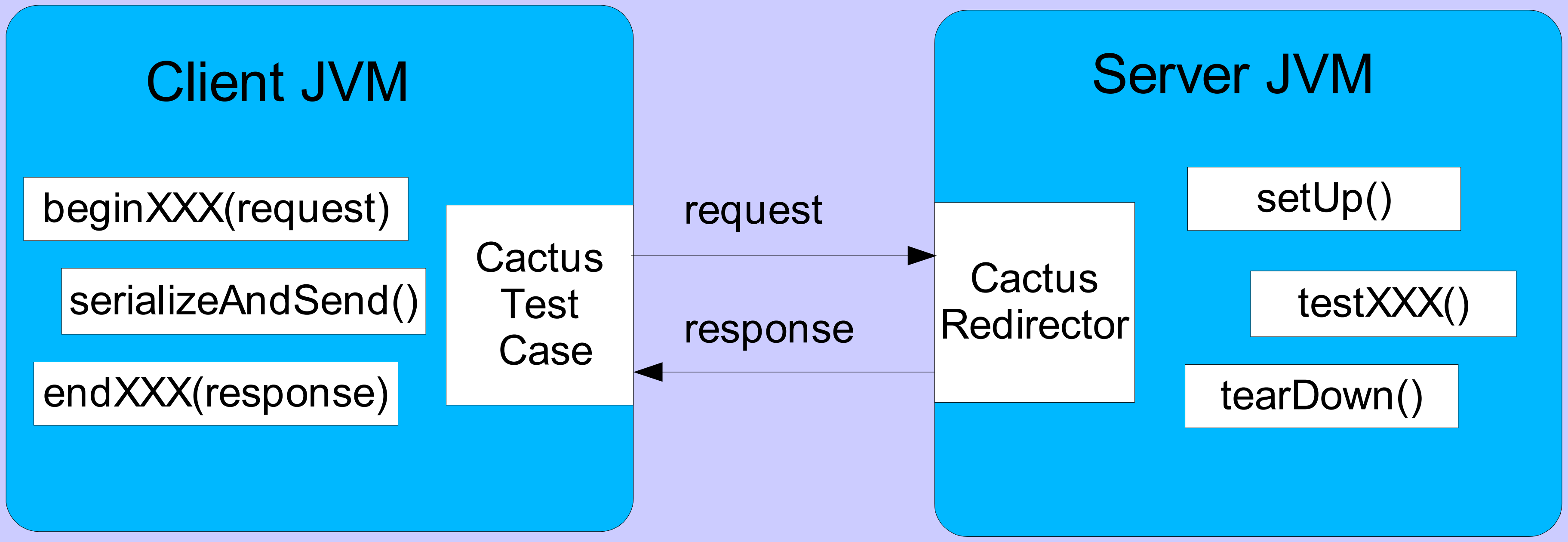
Where should you test?

- In your IDE
- In your Ant build (<junit> <junitreport> ...)
- Out of the container
  - Simplifies the test
  - Isolates the subject
  - May require use of Mock Objects for server-side objects
- In the container
  - Provides a truer *integration* test
  - Subject may not be *isolated*
  - Adds complexity in writing the test
  - May require container-specific modifications

Testing Server-Side Objects

- Cactus provides "in-container" testing
  - Extends JUnit (framework and <junit> Ant task)
  - "Out of the box" support for Tomcat, JBoss, Orion, Resin, and Weblogic
  - Includes Ant Tasks <cactifywar> and <cactus>
  - Provides access to *objectified* HTML response via HttpUnit integration
- Cactus can test Servlets, Filters, and JSPs, and EJBs
- Use StrutsTestCase for *Struts-aware* testing
  - As an extension to Cactus' ServletTestCase
  - Using mock objects for "out-of-container" testing

Cactus Test Architecture



1. *beginXXX* method is called to set up the web request

2. Cactus serializes the request and sends (using HttpClient)

8. *endXXX* method is called and passed web response

3. Redirector receives request

4. Calls *setUp*

5. Calls *testXXX* method

6. Calls *tearDown*

7. Cactus returns response

A Cactus Test Example

The Unit

/app/src/bank/GetAcctAction.java

```
package bank;
import org.apache.struts...
public class GetAcctAction
    extends Action {
 public ActionForward
      execute(...) {
  HttpSession sess = ...
  String id =
   request.getParameter(
     "id");

  Account acct =
    Account.load(id);

  sess.setAttribute("acct",
                   acct);
  return
   mapping.findForward(
      "success");
  }
}
```

The Unit Test

/app/test/src/bank/GetAcctActionTest.java

```
package bank;
import org.apache.cactus.*;
import junit.framework.*;
import org.apache.struts.action.ActionServlet;
public class GetAcctActionTest
           extends ServletTestCase {
  public void setUp() throws Exception {
    as = new ActionServlet();
    as.init(config); // required
  }
  public void beginSuccess(WebRequest req) {
    req.setURL(null,null,
      "/getAcct.do",null,"id=123");
  }
  public void testSuccess() throws Exception {
    as.doGet(request, response);
    assertNotNull("Acct stored in session",
      session.getAttribute("acct"));
  }
  public void endSuccess(WebResponse res){
  // verify forward ...}
  public void tearDown() throws Exception {
    as.destroy(); }
  private ActionServlet as;
}
```

The “Case” for StrutsTestCase

- Simplifies Testing Struts Actions
- Using Cactus alone requires knowledge of Struts' “inner workings”
- Extends Cactus and Junit – nothing special required in the build/test process
- Acts as a *base* class for testing your actions
- Built-in verifications (i.e. assertions) for Struts-specifics (action errors, forwards)
- Can be used in or out of the container

StrutsTestCase Example

The Unit

/app/src/bank/GetAcctAction.java

```
package bank;
import org.apache.struts...
public class GetAcctAction
    extends Action {
 public ActionForward
      execute(...) {
  HttpSession sess = ...
  String id =
   request.getParameter(
     “id”);

  Account acct =
    Account.load(id);

  sess.setAttribute(“acct”,
                    acct);
  return
   mapping.findForward(
      “success”);
  }
}
```

The Unit Test

/app/test/src/bank/GetAcctActionTest.java

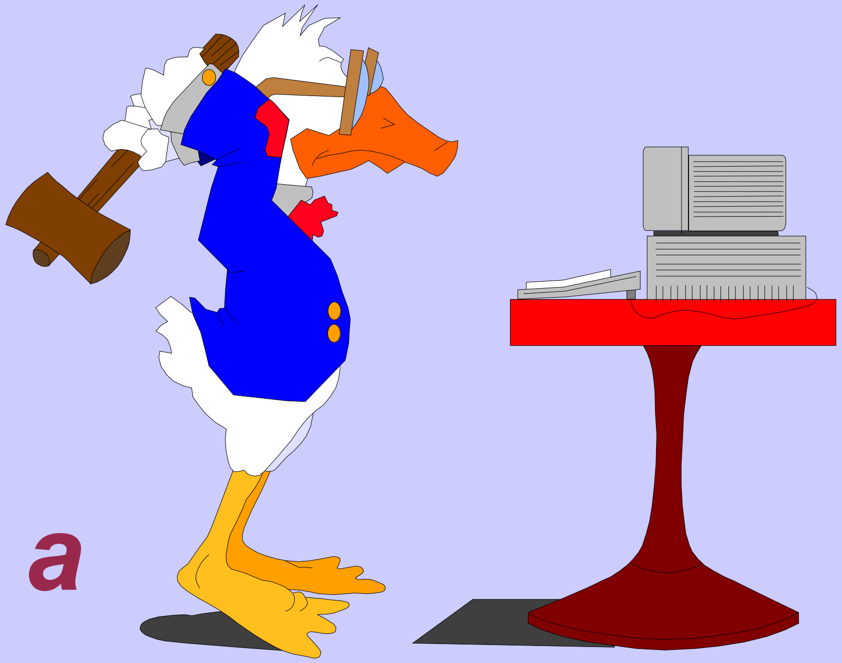
```
package bank;
import servletunit.struts.*;
import org.apache.cactus.*;
import junit.framework.*;
public class GetAcctActionTest
           extends CactusStrutsTestCase {
  public void testSuccess() throws Exception {
    setRequestPathInfo(“/getAcct”);
    addRequestParameter(“id”, “123”);
    actionPerform();
    assertNotNull(“Acct stored in session”,
      session.getAttribute(“acct”));
    verifyForward(“success”);
  }
}
```

Writing and Running a Cactus Test

- Cactus API
- Cactus Ant Tasks

Writing and Running a StrutsTestCase

- CactusStrutsTestCase
- MockStrutsTestCase



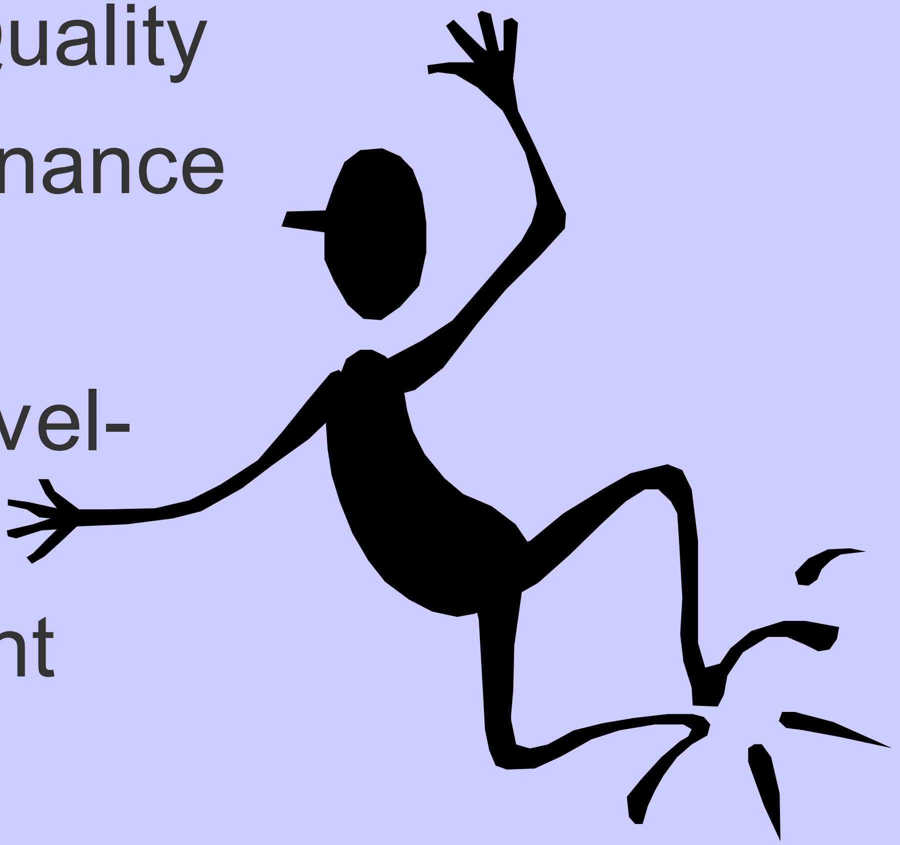
* Demo Time *

Struts Testing “Gotchas”

- Request Dispatcher forwards must be manually programmed in the test case.
- Cactus tests can significantly increase the build time.
- Mock test cases that rely on servlet filters or other container services may have problems.
- StrutsTestCase test cases do not have access to the web response (HTML).

Why are we testing anyway?

- Unit Testing Improves Quality
- Cost of Software Maintenance
- Cost of Bug Fixes
- A “Must” for Iterative Devel-opment
- Test-Driven Development
  - Improved API Design
  - A Driver for Refactoring
- Because it is FUN!

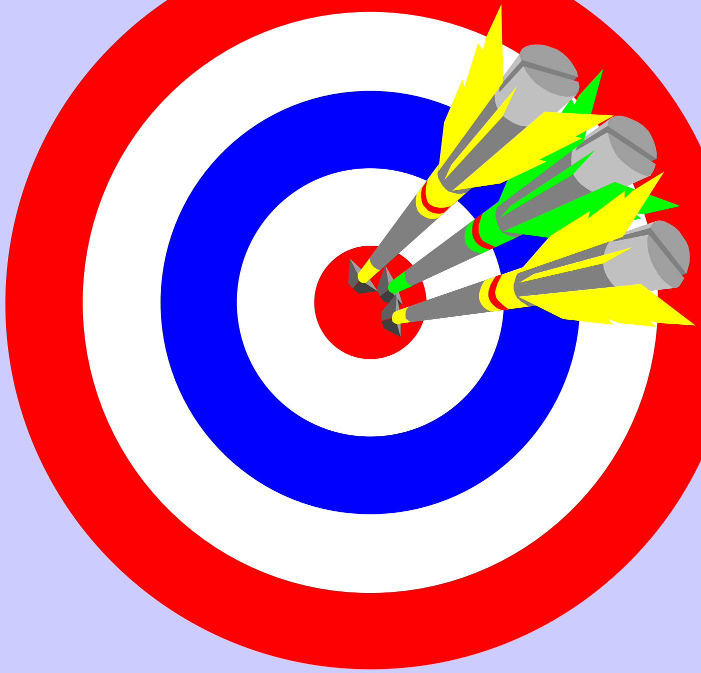


Best Practices of Testing

- Test Early – Test Often
- The quality of the unit being tested is only as good as the quality of the test itself!
- Use *setUp* and *tearDown* to ensure repeata-bility – a test method should be *idempotent*
- Leave external resources (database, file sys-tem, etc.) in original state
- Don't forget to test Exception Handling
- Never put *println* statements in a test method – Use Assertions Instead!
- Keep tests focused – Isolate the unit

Aiming for Test Quality

- Analyzing Test Coverage



* Demo Time *

Testing Frameworks / Resources

- JUnit (http://www.junit.org)
  - The Grandaddy of 'Em All
- Cactus (http://jakarta.apache.org/cactus)
  - Extends JUnit
- StrutsTestCase (http://strutstestcase.sourceforge.net)
  - Extends Cactus (or uses Mock Objects)
- HttpUnit (http://httpunit.sourceforge.net)
  - “Black-box” testing for web sites
  - Also has Simulated (mock) Servlet Container
- Clover (http://www.thecortex.net/clover)
  - Used to analyze coverage of Cactus itself
  - Free for open-source/non-commercial projects

Online Resources

- JUnit Testing Articles
  - http://junit.sourceforge.net/#Documentation
- Jakarta Pitfalls (Chapter 1)
  - http://www.theserverside.com/resources/articles/JakartaPitfalls/JakartaPitfallsChapter1.pdf
- Test flexibility with AspectJ and mock objects
  - http://www-106.ibm.com/developerworks/java/library/j-aspectj2
- Mock Objects http://www.mockobjects.com
- Interesting Google Searches
  - Java Test Coverage (Lots of hits!)
  - Java Unit Testing (Junit is the top hit)

Mission Accomplished!

Congratulations!

You have uncovered the benefits of Unit Testing! Time to pop the cork and celebrate your good fortune!