Functional Testing with Open Source Software

Quest 2009

Chris Kaufman

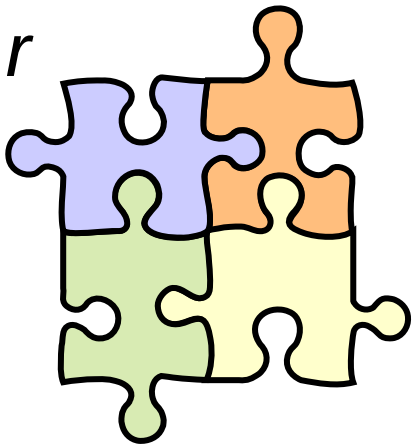
April 2009

Functional Testing of Equity Clearing System

- Testing began in February 2005 and is still being used
- Testing framework has since been reused on three other projects
- Framework components
  - Continuous build system
  - Automated build and test
  - Unit test with additions to support
    - Server-side in-container testing
    - Improved test reporting
    - XML verification

An Open Source Functional Testing Case Study

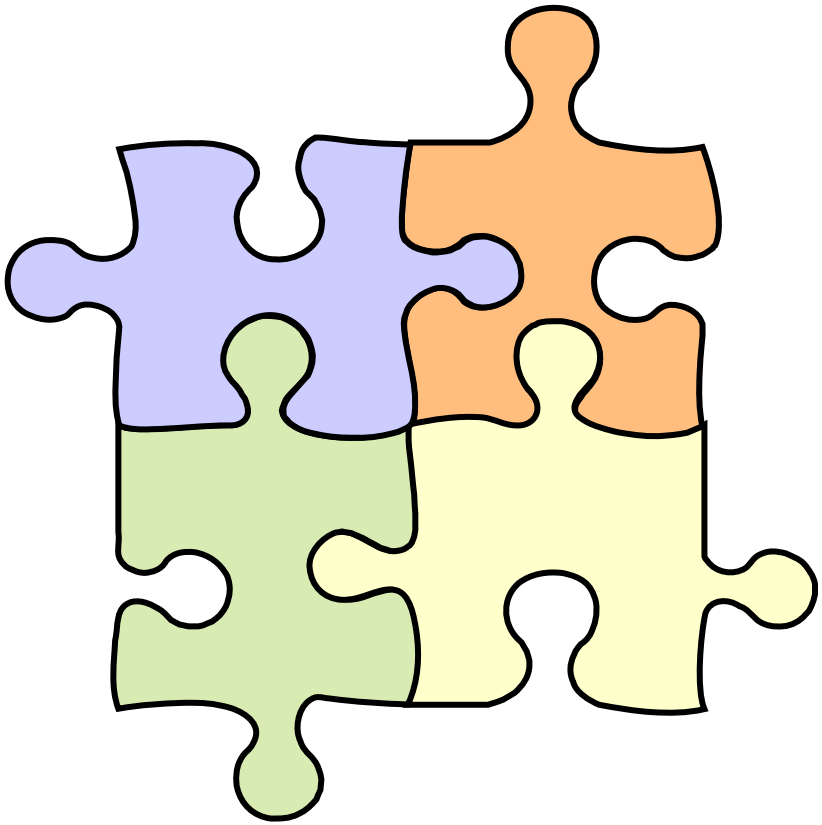
Putting the pieces together



- Background
- Testing Framework
- Using a Unit Test Tool for Functional Testing
  - Test Planning
  - Test Practice
  - Test Infrastructure
- Reflections

Project Background

Why try JUnit?



The Situation

- Our client was developing a Java-based clearing system for equities and derivatives
  - The system was based off of existing software which had some support for scripting which was used for legacy functional testing
    - No future development of scripting commands was planned
    - Scripting commands wouldn't cover any new functionality
  - The development team was adopting unit-testing for development of new tests
  - The client had no automated functional testing tools
  - The majority of the functionality was message based
    - Messages came in and out through queues implemented as database tables

The Situation (Continued)

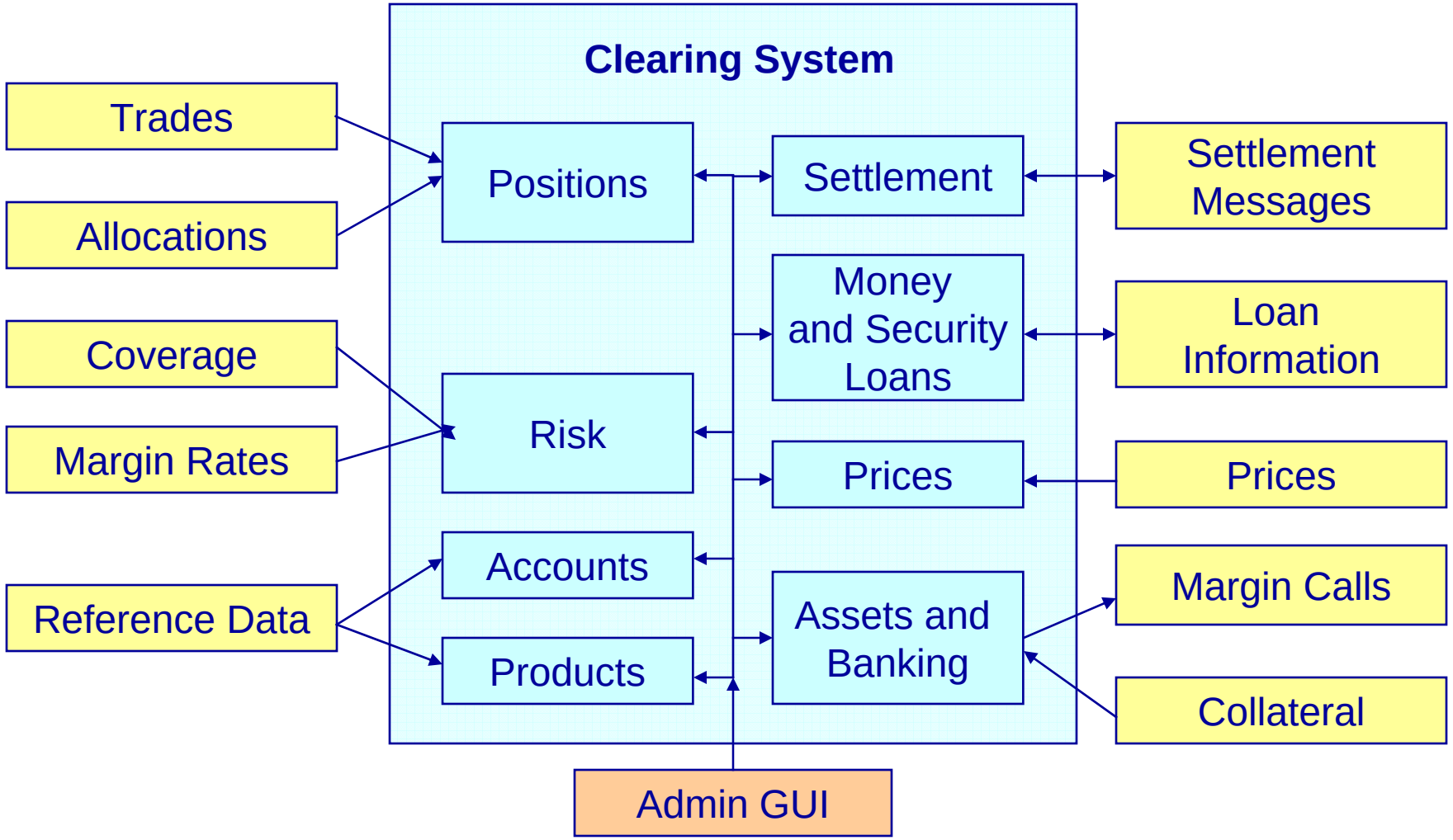
- Our client wanted an automated functional/regression test
  - That was cost effective
  - That could allow for development and test at multiple sites (a licensing consideration)
  - That could test the majority of end-to-end functionality, but did not necessarily have to test the GUI
    - The GUI was mostly used for reporting and to manually perform functionality that was also handled by external systems through the data interfaces
  - That could be implemented by a QA team with mixed technical skills

Technology

- Application Server
  - Windows 2003 Server (2008 Current)
  - BEA Weblogic 8.1 (9.2 Current)
  - Java 1.4 (1.5 Current)

- Database Server
  - Windows 2003 Server (2008 Current)
  - SQL Server 2000 (2005 Current)

System Block Diagram



The Question: Could JUnit Be Used?

- Although not designed for functional testing it provided a means to:
  - interact with the system
  - verify results of those interactions

Tool Analysis

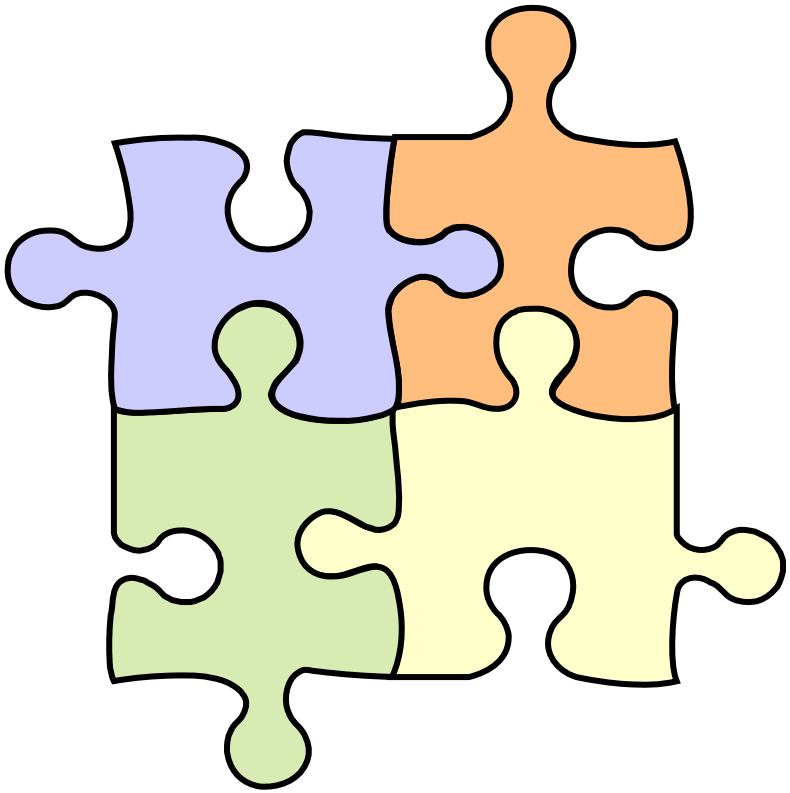
	OpenSource JUnit	Commercial Tool
Cost	Free – Unlimited licensing	Expensive –limited licensing
Learning curve	Harder, but developers can help	Simple Scripting
GUI testing	Possible with HttpUnit, HTMLUnit, etc.	Built in
Leverage existing script commands	Yes	Yes

Decision: Try JUnit

- Once the decision was made to try using JUnit we needed to
  - Determine how to use a unit test tool for functional testing
  - Determine how to structure our testing
  - Determine how to get the team members with no Java skills started with JUnit

Testing Framework

How the technology fits together



Tools

- CruiseControl – Continuous integration
- Apache Ant – Build tool
- JUnit – Unit test tool
- JUnitEE – J2EE JUnit test runner
- Cactus – Framework for server side unit testing
- XMLUnit – Extends JUnit with assertions for xml

CruiseControl

- CruiseControl is both a continuous integration tool and an extensible framework for creating a custom continuous build process.
- It includes dozens of plugins for a variety of source controls, build technologies, and notifications schemes including email and instant messaging.
- A web interface provides details of the current and previous builds.

Source: cruisecontrol.sourceforge.net

Apache Ant

- Apache Ant is a software tool for automating software build processes.
- Ant is implemented using the Java language, requires the Java platform, and is best suited to building Java projects.
- Ant uses XML to describe the build process and its dependencies.

Source: ant.apache.org

JUnit

- JUnit is a unit testing framework for the Java programming language.
- Created by Kent Beck and Erich Gamma, JUnit is one of the xUnit family of frameworks that originated with Kent Beck's SUnit (for Smalltalk)

- Source: www.JUnit.org

JUnitEE

- JUnitEE provides a TestRunner which outputs HTML and a servlet which can be used as an entry point to your test cases. Building your test harness as a standard J2EE web application means:
- Your tests are packaged conveniently into a .war file which can easily be moved between servers; you can leave the .war file in the main .ear file and simply avoid enabling the test web application on the production server.
- Your test classes will be dynamically reloaded by the app server (assuming your server supports this).
- Your test cases look just like your production code, and can use the same beans (or whatever) you use as a facade for your EJBs

Source: http://www.JUnitee.org/

Cactus

- Cactus is a simple test framework for unit testing server-side java code (Servlets, EJBs, Tag Libs, Filters, ...).
- The intent of Cactus is to lower the cost of writing tests for server-side code. It uses **JUnit** and extends it.
- Cactus implements an in-container strategy, meaning that tests are executed inside the container.

Source: jakarta.apache.org/

XMLUnit

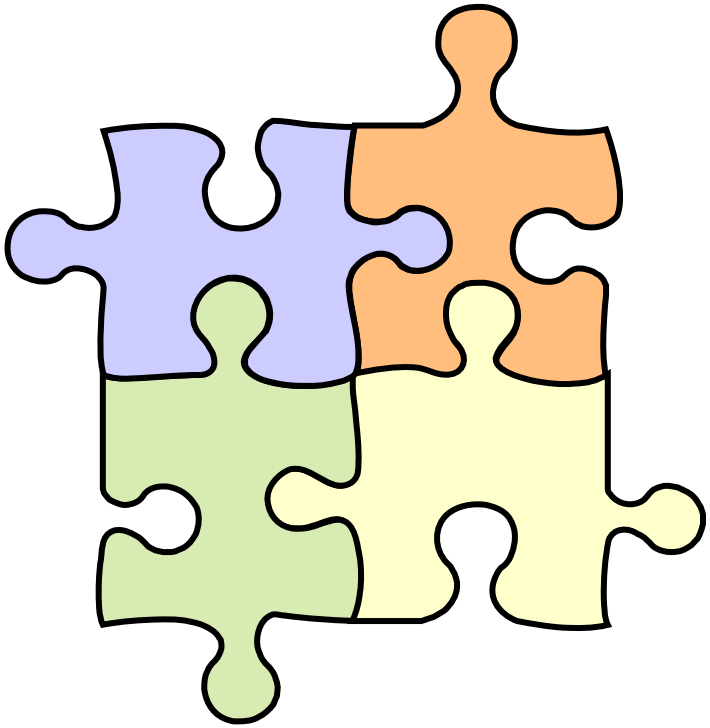
- XMLUnit allows assertions to be made about
  - The differences between two pieces of XML
  - The outcome of transforming a piece of XML using XSLT
  - The evaluation of an XPath expression on a piece of XML
  - The validity of a piece of XML
  - Individual nodes in a piece of XML that are exposed by DOM Traversal

Source: xmlunit.sourceforge.net

Using a Unit Test Tool for Functional Testing

How does the paradigm fit?

How does it need to be adjusted?



How to Functional Test with a Unit Test Tool

- Examine unit test principles
  - Decide how they apply to functional testing
  - Which can be kept?
  - Which must be modified or discarded?
  - What needs to be added to meet functional test needs?

Unit testing principles

- Test the smallest unit of functionality (in Java a method)
- Tests can be run independently
  - Each test does its own setup
  - Tests are isolated to the unit under test through stubs or mocks of other classes
  - Each test does its own teardown (returns the system to its starting state)
- Write test first before code

Unit Testing Principles Applied to Functional Test

- Test smallest unit of functionality
  - Tests should be as focused as possible.
- Make tests as independent as possible
  - Failure of any two tests should be as independent as possible.
    - Result of any test should not depend on other tests having run
    - This is constrained by time dependencies
    - Since these are not unit tests, there can be blocking defects
- Each test does its own setup
  - Test suite or test launcher does initial system setup
  - Each test does its own setup
  - Some setup can be shared

Unit Testing Principles Applied to Functional Test

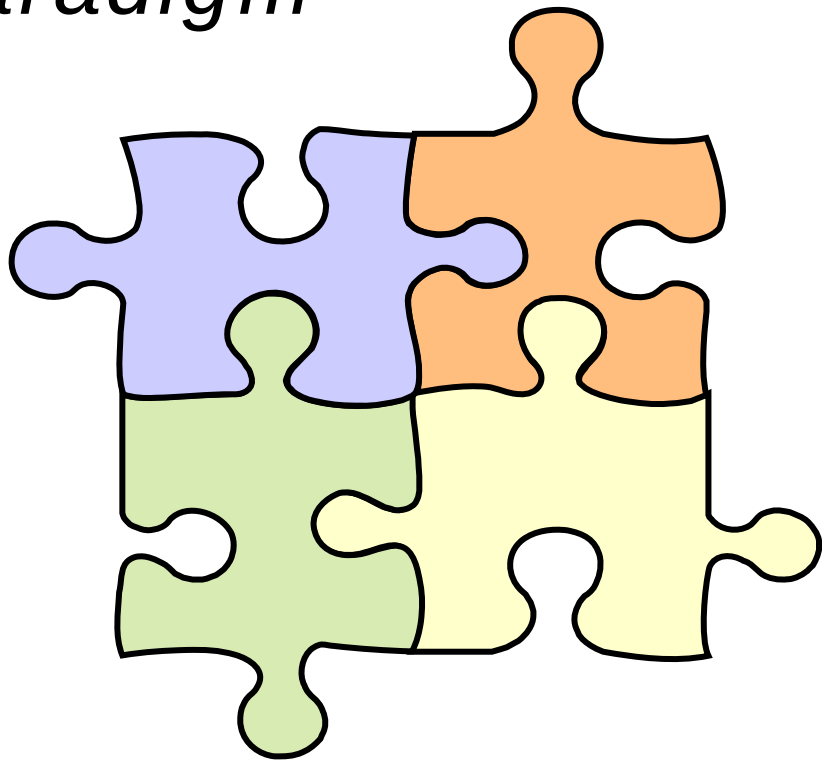
- Test teardown
  - Instead of returning system to initial state, return it to operating state.
  - As an example, this meant reopening any message gateways that had been closed due to errors.
- Write test first, then code
  - A significant chunk of development had been done before we started so we didn't even try this
  - Once we got to customer acceptance testing, we were able to write tests to reproduce customer reported defects before they were fixed by development.

Notable Differences

- With functional testing, some JUnit tests are not “tests” they are simply test “steps”
  - Setup – loading data
  - Processing – running an intermediate process which is not directly being tested
- Some JUnit tests may have more than one result
  - Each test can only report pass/fail
  - Error message can include multiple failure messages
  - Inside test JUnit asserts are caught, failure messages concatenated, and at end of test failure is thrown with list of failures

Test Planning

Fitting our test planning to the testing paradigm



Test Organization

- Assigned numeric range to test conditions for each function area. E.g.,
- 0001000-0001999 Position management
- 0002000-0002999 Account management
- 0003000-0003999 Product management
- 0004000-0004999 Settlement
- Etc.

Test Cases

- All test cases recorded in common format
- Automated tests incorporate test ID into data (account name, comments, ID, phone number, etc.) when ever possible. It makes tracking down problems easier.

Test ID	Test Condition	Test Day	Expected Result	Automated
0001000	Controlled Account trade of type X for current day	T	Trade accepted, position updated…	Yes
0001001	Controlled Account trade of type X for future date	T+1	Trade rejected, Error message…	No

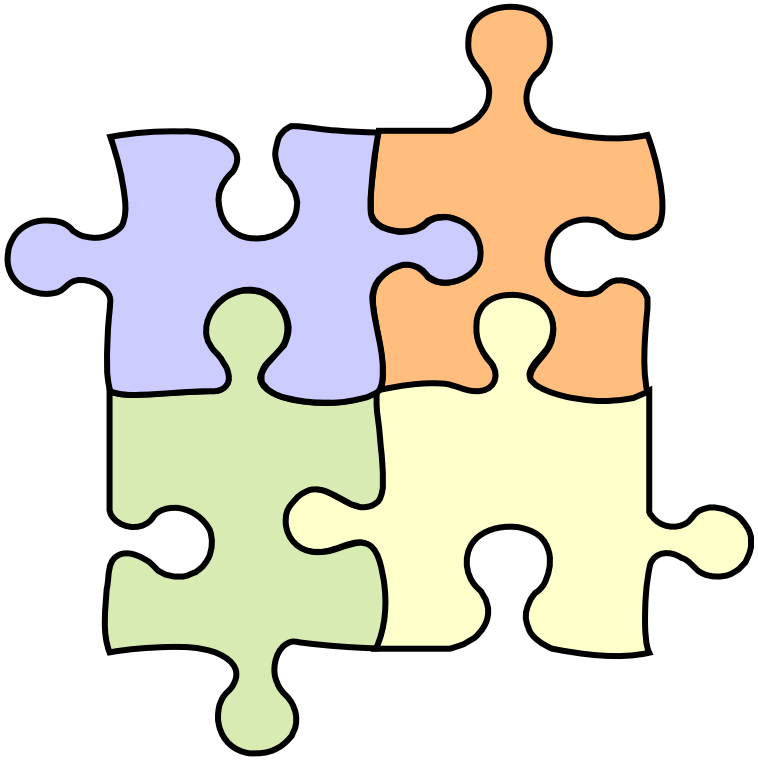
Test Data

- We created base test data for the system: Brokers, Accounts, Securities, Prices, Margin Rates, Users, etc.
- We used those only in tests where we didn't
  - change or delete the definition of the item
  - Need to keep and verify separate totals
- For each functional area we planned how we would separate data where we would make changes or verify totals
- Each functional area was assigned ranges of data to use.

Code and Data Files

- We elected to organize our tests under a separate directory under our project
- Tests were organized by functional area and named Sys0001001test.java
  - Where 0001001 is the test number
- Data files were organized by message type and named Sys0001001-1.xml
  - Where 0001001 is the test number
  - And –1 is the step of the test
- Verification files were named Verify0001001-n.xml following the same model as data

Making the tests work



General Form of Tests

- Simple Test
  - Load a message into a queue
  - Wait for message to be processed
  - Check database for expected result
- Most tests consisted of many messages
  - Create a broker
  - Create a trading account
  - Load a trade
  - Allocate the trade
  - Etc.
  - Check results

Achieving Test Independence

- Data independence
  - Structure your test data so that a failure of one test is less like to cause other tests to fail.
  - Tests involving calculations were separated by creating new accounts for each test.
- BUT, tests are related by time
  - In a equity clearing system there are events that happen on day T (the day of the trade), and on subsequent days, T+1, T+2, etc.
  - Tests that can happen on day T can be run independently of each other
  - Tests that run on day T+N rely on N-1 days of clearing being run prior to the test

Multi-day Tests

Some tests have setup, processing and verification that span multiple days of processing. In the basic stand-alone test all these steps are part of the same test. In the multi-test suites these had to be broken up into different tests.

Test1

- Test1 DayT Setup
- Day T End-of-Day processing
- ...
- Test1 DayT+N Setup
- Test1

Test2

- Test2 DayT Setup
- Day T End-of-Day processing
- ...
- Test2 DayT+N Setup
- Test2

Multi-Day Tests Combined

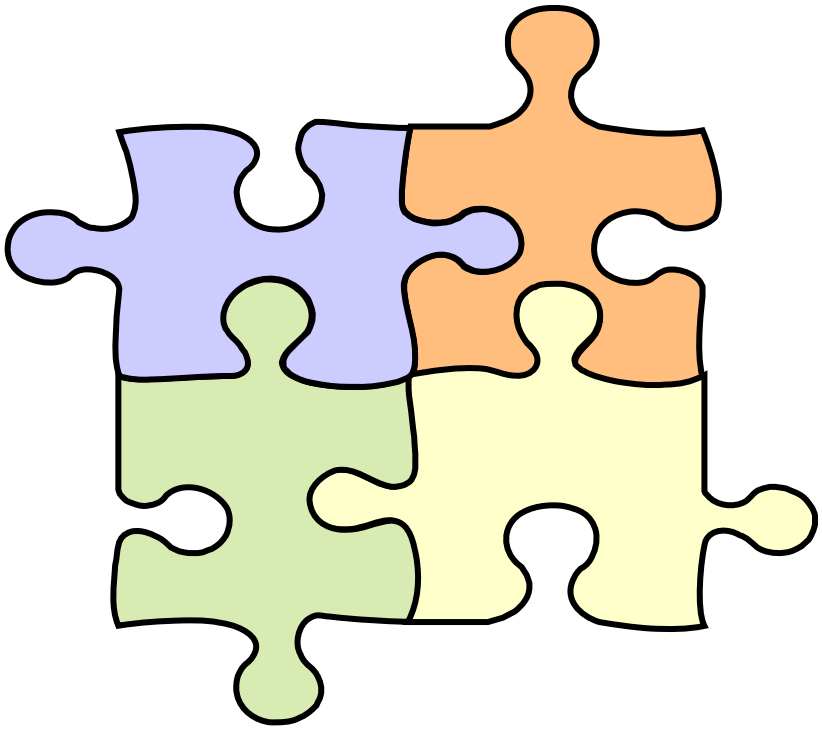
- Day T Setup
  - Test 1 Day T setup
  - Test 2 Day T setup
- Day T End-of-Day processing
- ...
- Day T+N Setup
  - Test 1 Day T+N setup
  - Test 2 Day T+N setup
- Test 1
- Test 2

Integration of Tests into Suites

- New tests were written stand-alone
- Once they passed, they were integrated into the regression test
- For each Day T+N test we setup two test cases
  - Test001001 and Test001001a
  - The “a” version of the test included all setup from prior days
- As the time grew to execute the regression suite we realized we needed a quicker way to validate builds, so we created a subset of tests that we used as a smoke suite.

Test Infrastructure

Beyond Open Source, what did we need to build



Test Utilities

Utilities were written to handle the most common tasks

- Load a message into a queue
- Check to see if a message was processed
- Verify database
- Perform functions usually handled through GUI
  - Run batch jobs
  - Open/close queues

Database Verification

- To simplify test writing for QA testers with limited programming experience we wrote a custom verify method that took one argument an xml file that described the verification to be done. That way our tests looked like.
  - LoadMessage(filename, messagequeue);
  - WaitForQueueEmpty(messagequeue);
  - VerifyDatabase(filename)

Database Verification

File consisted of query and results, with columns pipe delimited. Multiple rows could be returned, but an order by clause was needed on sql for repeatable results

```
<?xml 1.0>
<verify
    sql="select col1, col2, col3 from table1 where
      col4=expectedvalue order by col1">
    <result>val1|val2|val3</result>
    <result>valx|valy|valz</result>
</verify>
```

Database Verification

Data from one database table could also be verified against another by running two queries and comparing results.

```
<? xml 1.0>
<verifydb
    sql1=“select this, that from table1”
    sql2=“select this, other from table2” />
```

File Verification

We also wrote a generic file verification that would compare a file against expected results. The xml control file included or exclude rows based on regular expressions, and transformed the resulting rows based on a pattern match

```
<? xml 1.0>
<filecompare filename="filename">
  <include>regexp</include>
   <exclude>regexp</include>
  <match>regexp<match>
   <result>1234abc90</result>
   <result>3456xyz89</result>
</filecompare>
```

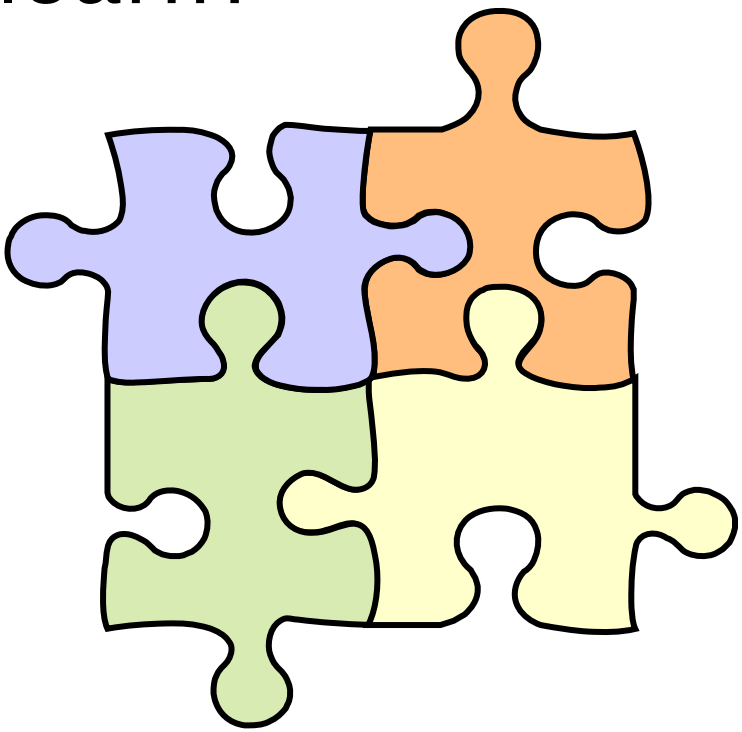
Developing Tests Without (New) Coding

- Once we had our utilities complete we could write the majority of our test with boilerplate code.
- The majority of the effort was spent developing test message files, and test verification files.
- The members of our team with coding experience took on tasks where new utilities needed to be developed.
- The rest of the team worked on the bulk of the testing which concentrated on data and results

Reflections

How did it turn out?

What did we learn?



Planned Benefits

- The test framework and verification methods were ported to several other projects
- We could refactor software with confidence that we could be assured it still functioned properly
- HTTPUnit was added for verification of GUI screens on one of those projects
  - The particular mix of tools we had did not let us use HTMLUnit or other higher level unit test

Unexpected Benefits

- Our messages that were output to other systems were also xml messages. When we started verifying them we looked to see if there was a unit test tool for xml.
  - There was, it was called XMLTest
  - In less than two days we integrated it and could begin writing verification tests for tags or attributes in an xml message

Unexpected Benefits

- When we needed to do some performance testing, we already had a framework for inserting messages into queues. It was relatively simple to create a test case that read files from a directory and put them in the queue based either on a fixed messages per second or a min and max number of unprocessed messages to allow in the queue.
- This unit test could be run on the same or a different machine.

Robustness of Solution

- Original Application being tested ran on:
  - Java 1.4.2
  - Weblogic 8.1
  - SQL Server 2000
- In 2008 it was ported to
  - Java 1.5
  - Weblogic 9.2
  - SQL Server 2005

After upgrading to a newer version of JUnitEE everything worked, except some expected results had to be changed because of how zero values were returned from the database.

Problems Unique to Open Source

- Most open source packages are built on other open source packages
  - We encountered problems with incompatible versions of common jars
  - The impact
    - We could use HTTPUnit to write low-level tests
    - But HTMLUnit or
    - Canoo failed.
  - When we upgraded to Java 1.5 and new versions of all our test software we lost the ability to nicely format in html our test results from JUnitEE
- The problems were no worse or limiting than problems I have experienced with commercial software

Future Directions

- Some ideas of how our framework could be improved
  - Distinguish between setup and verification tests for reporting
  - Fail an entire test run under certain conditions
    - If end-of-day processing fails, all subsequent tests are suspect
  - Smart test-runner framework
    - Each test could include prerequisite steps
    - Each test could include constraints (cannot run after)
    - User could select a handful of tests and all required setup and prerequisite steps would be scheduled and run