

**A random sample of 25 slides from
“*Agile Testing for Java Developers*”
course**



JUnit terminology

- Test case – A test on a single function of a class under test
- Test suite – A collection of tests or suites
 - Recursive
- Failure – Anticipated problems
 - Conditions that are checked against expected values
- Error – Unanticipated problems
 - Exceptions
- Fixture – A place to hold class(s) under test
- Assert – Check for an expected condition

Important JUnit classes

- **junit.framework.Test**
 - Interface that describes an object that can be a test
- **junit.framework.TestCase** (implements **junit.framework.Test** interface)
 - Provides test behavior such as assertions, logging
 - Subclass this for your own test classes
- **junit.framework.TestSuite** (implements **junit.framework.Test** interface)
 - Collects and runs tests. Can hold instances of TestCase or TestSuite
- **junit.framework.TestResult**
 - Collects results of running a test
 - Usually rendered by test runner
 - Pass this to a TestCase to run it
 - `testCase.run(TestResult)`
 - Test runner does this for you

Writing tests with JUnit

1. Create test class
2. Add test method(s)
3. Optionally use test fixture instance variables
4. Optionally use `setUp()` and `tearDown()` for initialization and cleanup
5. Optionally create test suites to organize multiple tests

Collecting tests

- JUnit uses reflection to locate all test methods within a `TestCase` class
- JUnit looks for all methods that fit the “`public void testxxx()`” pattern
- You can override this behavior in the `suite()` method
- Collect a subset of tests in a test class
- Collect tests from more than one test class

jcoverage and jcoverage++

- Adds instrumentation directly to compiled class (byte code insertion)
 - Runs a little faster than JPTA-based tools
- Available from <http://www.jcoverage.com/>
 - jcoverage is open source (GNU public license)
 - jcoverage++ includes other tools and costs ~ \$1k/developer/year with volume discounts
- Use it through ANT
 - Provides tasks for to
 - Instrument code
 - Run test
 - Merge test runs
- Also has command line interface

Instrumenting code with jcoverage

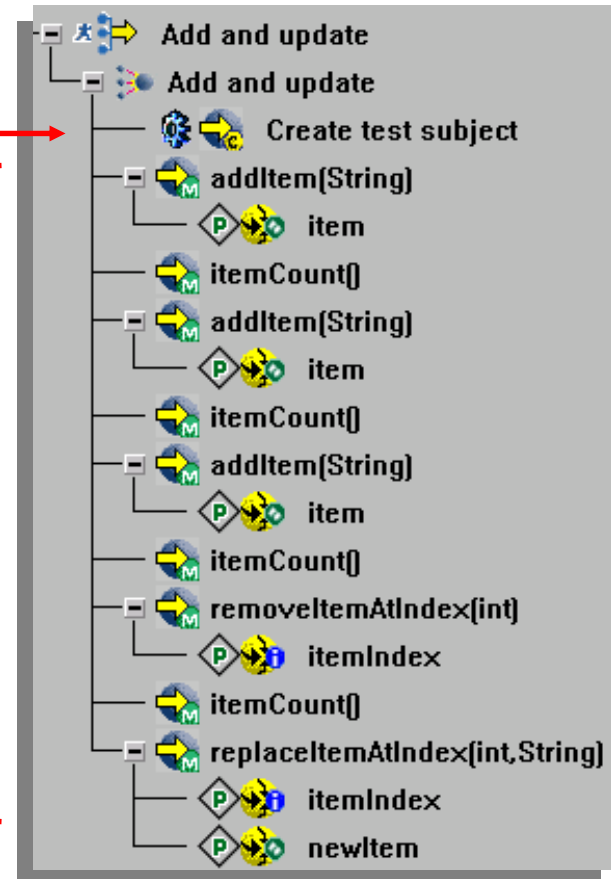
- Use `<instrument>` task
- Prerequisites:
 - This classpath *must* include the location of the `jcoverage.jar` or `jcoverage-plus.jar`
 - If the classes were compiled with Ant, the `javac` task should be specified with `debug="true"`
- Generates new versions of the classes, with extra monitoring bytecodes inserted
- Generates `jcoverage.ser` file that contains an entry for every source line, each initialized with a hit count of 0

Instrumentation example

```
<instrument todir="build/instrumented-classes">  
  <fileset dir="build/classes">  
    <include name="**/*.class"/>  
  </fileset>  
</instrument>
```

Generated test (visual style)

- Generated test asset
- Creates test subject
- Creates a method step for each recorded interaction
- Delete ones you don't need
- Add postcondition validation as necessary



Factors that effect testability

- To test a component, you:
 - must be able to control it
 - controllability
 - must be able to observe its output or state
 - observability

Architectural considerations

- A highly fault-tolerant system inhibits testability by reducing the ability to observe defects
 - Such as catching exceptions at too low a level
- Concurrency inhibits testability by reducing controllability through timing constraints and temporal non-determinism
- Classes may have dependencies on supporting classes
 - supporting classes are incomplete
 - supporting classes are difficult to set up or control
- Hiding internal state via private fields inhibits observability

Types of FIT fixtures

- Column Fixture – sets up given fixture with test values from a table row and validates that a one or more methods called return the correct values
- Action Fixture – executes a sequence of actions on a component
 - Used for implementing *action words* tests
- Row Fixture – Used to validate that a collection of components is in an expected state

Action Fixture

- Action fixture interprets rows in a table as a sequence of commands to be performed in order
- The first column represents the command to execute
- The remaining columns represent parameters to be passed to the action

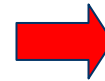
Default actions

- `fit.ActionFixture` supplies the following default actions.
 - **start** `aClass` -- Subsequent commands are directed to an instance of `aClass`. This is similar to navigating to a particular GUI screen.
 - **enter** `aMethod` `anArgument` -- Invoke `aMethod` with `anArgument` (of type determined by `aMethod`.) This is similar to entering values into GUI fields.
 - **press** `aMethod` -- Invoke `aMethod` with no arguments. This is similar to pressing a GUI button.
 - **check** `aMethod` `aValue` -- Invoke `aMethod` with no arguments. Compare the returned value with `aValue`. This is similar to reading values from a GUI screen.
- Subclass `fit.ActionFixture` to add your own actions.

Action fixture example

Before test run

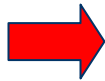
<u>fit.ActionFixture</u>		
start	eg.music.Browser	
enter	library	Source/eg/music/Music.txt
check	total songs	37



After test run

<u>fit.ActionFixture</u>		
start	eg.music.Browser	
enter	library	Source/eg/music/Music.txt
check	total songs	37

<u>fit.ActionFixture</u>		
start	eg.music.Browser	
enter	library	Source/eg/music/Music.txt
check	total songs	36



<u>fit.ActionFixture</u>		
start	eg.music.Browser	
enter	library	Source/eg/music/Music.txt
check	total songs	36 <i>expected</i>
		37 <i>actual</i>

To add more actions, simply subclass fit.ActionFixture and add a method for each new type of action

Fixture tips

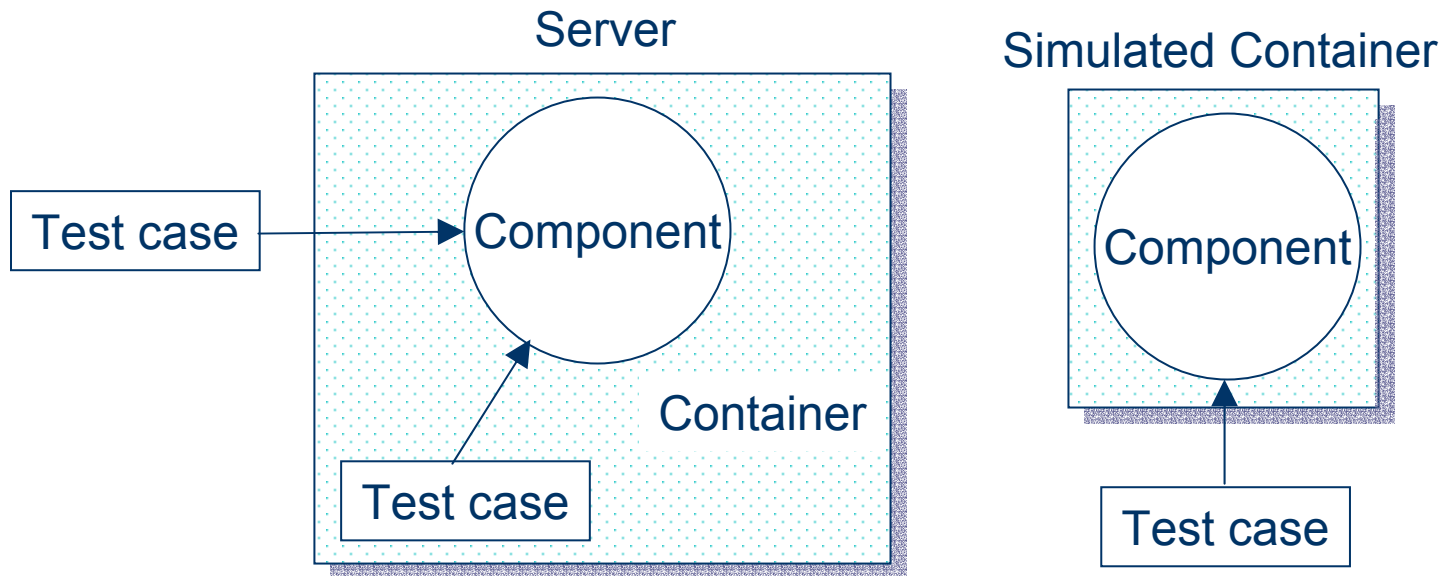
- Remove surplus reporting in `RowFixture.doRows (Parse)` via subclass if you want to validate only a subset of rows returned
 - An example of this is provided in `agile.tests.SurplusRowIgnoringRowFixture`
- Use static variables in fixtures to hold objects for other fixtures to use
 - For example, create and configure an object using an `ActionFixture`. Use a static to hold object so `RowFixture` can query it.

State extraction

- Specifies means to describe a particular view of the aggregate state of an object
- Uses a set of “getter” methods and fields that will extract just those values that are of interest
- May walk tree of contained objects in order to return “flattened” list of primitive states
- Used for validation

Server testing choices

- Run tests as clients of server components
- Run tests directly on the server itself
- Simulate server container



Running Cactus

- Three ways to run Cactus
- Run from Ant
 - Use `<cactus>` task
 - Launches a new container each time
- Run from JUnit runner
 - Must have container already running Cactus
- Run from browser
 - Cactus provides a servlet that will execute a specified test suite
 - `http://localhost:8080/test/ServletTestRunner?suite=TestSampleServlet&xsl=junit-noframes.xsl`
 - Must have container already running Cactus

Calling servlets with ServletUnit example #1

```
public void testServlet() {
    ServletRunner sr = new ServletRunner();
    sr.registerServlet( "myServlet", StatefulServlet.class.getName() );
    ServletUnitClient sc = sr.newClient();
    WebRequest request = new PostMethodWebRequest(
        "http://www.mycompany.com/myServlet" );
    request.setParameter( "color", "red" );

    WebResponse response = sc.getResponse( request );
    assertNotNull( "No response received", response );
    assertEquals( "content type",
        "text/plain",
        response.getContentType() );
    assertEquals( "requested resource",
        "You selected red",
        response.getText() );
}
```

Running SQLUnit from ANT

2. Run the tests

```
<target name="run-all-tests" depends="compile">
  <sqlunit haltOnFailure="false" debug="false">
    <fileset dir="stored_procedure_tests">
      <include name="**/*.xml" />
    </fileset>
  </sqlunit>
</target>
```

Assignment

- Use JUnit to test `agile.LoanCalculator*`
 - Create an instance of `agile.LoanCalculator` with
 - Principal = 250000, term = 30, rate = 6.5
 - Use JUnit assertion APIs to validate that:
 - Monthly payment = 1580.17
 - Number of payments = 360
 - Total principal paid = 250000
 - Total interest paid = 318861.22
- * See appendix for description of `agile.LoanCalculator`

Assignment

- Use test-driven development and JUnit to design and develop a ToDoList class according to the following user stories:
 - Add an item to an empty list, make sure it is added
 - Add an item, make sure it is added to the end of the list
 - Remove an item at an index, make sure it is removed
 - Replace an item at an index with a new item , make sure it is replaced
 - Print a formatted report of the items
- Determine a reasonable set of data variations for each story
- Make sure you create the tests first!

Example object to mock

- We will mock a communications channel object
- **Channel** public interface:
 - `void write(Packet) throws WriteException`
 - `void setClient(Listener)`
 - `Packet read() throws ReadException`
- A **Packet** is an object containing a header and between 0 and `MAX_PACKET_DATA` bytes of data
- One or more packets may need to be sent in order to transmit a complete message
 - It is assumed that the receiver will figure out which packets belong together

Assignment

1. Automate the build for the JUnit sample tests using ANT
 - JUnit samples source files are in `c:\junit3.8.1\junit\samples`
 - Place all compiled.class files in `junit_samples.jar`
2. Add a task to run the JUnit sample tests in `junit_samples.jar` within the ANT script above

Assignment

- Use `jcoverage` in the JUnit samples Ant build file you created to run and measure coverage on the JUnit samples
- Hints
 - This classpath *must* include the location of the `jcoverage.jar` or `jcoverage-plus.jar`
 - The `javac` task should be specified with `debug="true"`