JRunit Guide

2005, Tom Elrod and Clebert Suconic at JBoss Inc

Table of Contents

1

Overview - What is JRunit

JRunit is a project to aid in adding benchmarking to JUnit based test cases as well as providing a framework extension to JUnit to allow for distributed client/server based tests. It is important to note that JRunit is not a replacement for the popular test framework JUnit, but an extention to it allowing for more enterprise focused features.

With regards to client/server based testing, there are certain limitations of JUnit that make it impractical to use out of the box. The biggest limitation is that it was designed to have all tests run within a single process within a single class. It is also designed with the constraint that all tests are atomic and independent of any other code outside the test being run and that the lifecycle for these tests are independent for each test. This is all great for testing low level units of code, but not for client/server scenario tests.

Due to the specific nature of client/server code, there are some extra requirements for testing. The first is that the lifecycle, or state, of the client and server tests can be managed. This is important because will need to make sure that the server is created and initialized before any clients begin their tests which will call on the server and that the server does not shutdown before all the clients have finished their tests. Also need the ability to consolidate the results from all the client and server tests into a single result format. Finally, need to be able to have all this driven from a single point so can be included within an automated test run from a build.

JRunit addresses these issues by allowing multiple test cases, client and server, to be run simultaneously in different processes, but controlled through a central driver. This driver controls the complete lifecycle of all the test cases. Each of the test cases spawned by the driver are run within a harness that is responsible for reporting all the results back to the driver. This driver is itself a TestCase?, so can be run directly from an IDE or ant build as a JUnit test case.

JRunit also allows the addition of benchmark hooks into test code. These benchmark hooks will allow the JRunit framework to collect statistics on how long it takes to execute different sections of code. These statistics can be captured to a number of persistent stores and viewed as immediate results or captured over time to view how performance is impacted over time due to code changes.

2

How JRunit works

Client/Server tests

The first issue to tackle is the lifecycle of the server tests, since it will need to break from the default behavior of JUnit's lifecycle management.

To start, need a way to tell the server to start up and initialize. This should be done using `setUp()` method. Once this method returns, it is assumed that the server is ready to receive calls. Next, need a way to tell the server that it can shutdown and clean up. This should be done using the `tearDown()` method. This method will only be called in the case that all clients are finished making their calls to the server.

This behavior is all provided by having the server test case class extend `org.jboss.jrunit.ServerTestCase` instead of `junit.framework.TestCase`. The `ServerTestCase` actually extends the `TestCase` class itself, but overrides a few it's methods to get the desired behavior.

The server implementation may also have a test method, which will be called just after the `setUp()`, as in the case of regular JUnit test runs. This test method can contain asserts and are suggested to be used for validation of server data and metrics.

A few important points… there can be only *ONE* test method within a server test case. This is because JUnit creates a new instance of the test class for each test method run. Therefore, if had multiple tests, would be multiple instances of the server test case created. To change this would require a sizable change to the JUnit code, so please just use one test method (or contribute the required change #).

Another important point is that the `tearDown()` method may be called even while a test method is being run. This is intentional and allows for the test method to loop until the `tearDown()` method is called. So a possible example of where this could be used is:

```
...
   public void testServerMetrics()
   {
      while(!stop)
      {
         // collect data here
      }
   }

   protected void tearDown()
   {
      stop = true;
      // so will cause testServerMetrics() to break out of loop
      // do shutdown and clean up code.
   }
...
```

For the client test case, there is no requirement for jrunit other than they extend the `junit.framework.TestCase` class and conform to normal constraints of a JUnit test case.

Lastly, there is the `org.jboss.jrunit.TestDriver` class, which represents the driver for the client and server tests. This class will spawn new test harnesses that the client and server tests cases will run within. The test driver will then communicate to the test harnesses using a JGroups message bus to control the test lifecycle for the server test case and all the client tests cases as well as obtaining the results from those test runs.

The logical order of a test run as controlled by the test runner is:

1. Spawn a new test harness process for each client and server test case.
2. Wait for confirmation that all test harnesses have been created and their message bus has been started.
3. Once confirmation has been received, wait for server test case to start up (i.e. call the setUp() method on the server test case). Otherwise, if confirmation not received, kill all the processes and return error to JUnit.
4. Once the confirmation of the server startup has been received, tell all the test cases to run (client and server). Otherwise, if confirmation not received, send abort message to all the test harnesses.
5. Wait for results from all the client test cases.
6. Once all the client test results are received, tell the server to tear down (i.e. call the tearDown() method on the server test case). Otherwise, if results not received, kill all the processes and return error to JUnit.
7. Wait for server test results (if server test case had a test method).
8. Process all results by adding them to root JUnit TestResult?, which will be reported via normal JUnit reporting.
9. Wait for server torn down message, indicating it has successfully cleaned up.
10. Shutdown message bus and end root test run, returning JUnit execution thread.

The only user coding required for the test driver is to implement a class that extends the `org.jboss.jrunit.TestDriver` abstract class and implement the `declareTestClasses()` method. Within this method, call the `TestDriver`'s `addTestClasses()` method and specify the client test case class, the number of clients to spawn, and the server test case class.

Benchmark Decorators

JRunit can also utilize the benchmark decorators to provide benchmark results as well as regular JUnit test results.

`org.jboss.jrunit.decorators.ThreadLocalDecorator` (and all the other decorators that accept number of threads and loops) - for each thread specified in the number of threads, there will be a new instance of the test created. For the number of loops specified, the test methods will be called on each test instance. So for example, if specify 3 threads and 10 loops, there will be three instances of the test class created and each instance will have their test methods called 10 times by each of their respective threads. An example class that demonstrates this would be:

```
public class SimpleThreadLoopCounter extends TestCase
```

```
{
   private static int staticCounter = 0;
   private static int staticMethodCounter = 0;
   private int localCounter = 0;

   public static Test suite()
   {
      return new ThreadLocalDecorator(SimpleThreadLoopCounter.class, 3, 10, 0, true, true);
   }

   public SimpleThreadLoopCounter()
   {
      staticCounter++;
   }

   public void testCounter() throws Exception
   {
      System.out.println("staticCounter = " + staticCounter);
      System.out.println("staticMethodcounter = " + ++staticMethodCounter);
      System.out.println("localCounter = " + ++localCounter);
   }
}
```

When this is run, the last entry will be:

```
staticCounter = 3 staticMethodcounter = 30 localCounter = 10
```

Another issue is the way junit treats test classes with multiple test methods. For each test method that junit runs within a test class, it will create a new test instance to run that test method. To illustrate this, if copied the testCounter() method from the example above (calling it `testCounter2()` for example), the last lines printed for the test run would be:

```
staticCounter = 6 staticMethodcounter = 60 localCounter = 10
```

Issues with jrunit and decorators: The `ServerTestHarness` will not work if the `numberOfThreads` is more than 1. I don't see this as being that big of a problem from the point of view that the `numberOfThreads` is to simulate multiple clients running at the same time. If want to do this using the `ServerTestHarness`, can actually spawn multiple clients through it. Eventhough they won't be running concurrently with the same processes, they will be running concurrently with seperate processes. Having the ability to use the loop parameter is needed though so can keep the clients calling on the server for an extended period of time (or iterations). Therefore, if running remote tests and want to use the `ThreadLocalDecorator` for benchmark data, use the following `ThreadLocalDecorator` constructor, which will default the number of threads to one:

```
public ThreadLocalDecorator(Class testClazz, int loops)
```

3

Samples

This covers the samples include within the JRunit project. The main focus of the samples covered here are related to client/server tests. All the samples illustrate the different features of microbenchmark and jrunit using the same base client and server base code which uses a socket to send a String to the server from the client, which is then returned a static String from the server. All the samples can be compiled and run within an IDE, but some can be run via the ant build script included in the root directory of jrunit. The reason that only some are included within the ant script is that they can be completely automated (which will be covered below). Also note that each section title for the samples below corresponds to the org.jboss.jrunit.sample package that they can be found under.

basic

The first sample for client/server testing is in the `org.jboss.jrunit.sample.basic` package, which contains `SimpleServerTest` and `SimpleServerTest`. These are the tests that we will build on for the rest of the test cases that demonstrate the different features. The `SimpleClientTest` is a basic JUnit test case with two test methods, `testClientCall()` and `testFailure()`.

```
public class SimpleClientTest extends TestCase
{
   private String address = "localhost";
   private int port = 6700;
   private Socket socket = null;
   private ObjectOutputStream oos;
   private ObjectInputStream objInputStream;

   public void testClientCall() throws Exception
   {
      getSocket();

      oos.writeObject("This is the request from " + Thread.currentThread().getName());
      oos.reset();
      oos.writeObject(Boolean.TRUE);
      oos.flush();
      oos.reset();

      Object obj = objInputStream.readObject();
      objInputStream.readObject();

      assertEquals("This is response.", obj);

   }

   public void testFailure()
   {
      fail("This is a check to make sure failures reported.");
   }

   private void getSocket() throws IOException
   {
      if(socket == null)
      {
```

```
         try
         {
            socket = new Socket(address, port);
            BufferedOutputStream out = new BufferedOutputStream(socket.getOutputStream());
            BufferedInputStream in = new BufferedInputStream(socket.getInputStream());

            oos = new ObjectOutputStream(out);
            objInputStream = new ObjectInputStream(in);
         }
         catch(IOException e)
         {
            e.printStackTrace();
         }
      }
      else
      {
         oos.reset();
         oos.writeByte(1);
         oos.flush();
         oos.reset();
         objInputStream.readByte();
      }
   }
}
```

The `testClientCall()` method will open a socket to the server, send a String and receive a String in return and check that the returned String is the value it expected (using an JUnit assert). The `testFailure()` method will just fail a JUnit assertion.

The `SimpleServerTest` is a basic JUnit test case that sets up a socket server within it's `setUp()` method (on a separate thread) which will accept requests from clients and respond with a static String. The `SimpleServerTest` also has a test method, called `testRequestCount()`, which will simply print out the number of request its taken every 3 seconds.

```
public class SimpleServerTest extends TestCase //ServerTestCase
{

   private int serverBindPort = 6700;
   private int backlog = 200;
   private ServerSocket serverSocket;
   private InetAddress bindAddress;
   private String serverBindAddress = "localhost";
   private int timeout = 5000; // 5 seconds.

   private boolean continueToRun = true; // flag to keep the server listening

   private int requestCounter = 0;

   protected void setUp() throws Exception
   {
      System.out.println("SimpleServerTest::setUp() called.");
      bindAddress = InetAddress.getByName(serverBindAddress);
      serverSocket = new ServerSocket(serverBindPort, backlog, bindAddress);

      // this was done inline since TestCase already has a void parameter run() method
      // so could not create a run() method for the Runnable implementation.
      new Thread()
      {
         public void run()
         {
            try
            {
               startServer();
            }
```

```
            catch(Exception e)
            {
               e.printStackTrace();
            }
         }
      }.start();
   }

   public void testRequestCount()
   {
      while(continueToRun)
      {
         try
         {
            Thread.currentThread().sleep(3000);
         }
         catch(InterruptedException e)
         {
            e.printStackTrace();
         }
         System.out.println("Requests taken: " + requestCounter);
      }

      assertEquals(30, requestCounter);
   }

   protected void tearDown() throws Exception
   {
      continueToRun = false;

      System.out.println("Tearing down.  Processed " + requestCounter + " requests");

      if(serverSocket != null && !serverSocket.isClosed())
      {
         serverSocket.close();
         serverSocket = null;
      }
   }

   private void startServer()
   {
      while(continueToRun)
      {
         try
         {
            Socket socket = serverSocket.accept();
            socket.setSoTimeout(timeout);

            BufferedOutputStream bos = new BufferedOutputStream(socket.getOutputStream());
            BufferedInputStream bis = new BufferedInputStream(socket.getInputStream());

            ObjectOutputStream oos = new ObjectOutputStream(bos);
            oos.flush();
            ObjectInputStream objInputStream = new ObjectInputStream(bis);

            processRequest(objInputStream, oos);

            while(continueToRun)
            {
               acknowledge(objInputStream, oos);
               processRequest(objInputStream, oos);
            }
         }
         catch(IOException e)
         {
            System.out.println("Done processing on client socket.");
         }
```

```
         catch(ClassNotFoundException e)
         {
         }

      }
   }

...
```

This introduces one of the problems with JUnit testing from a client/server standpoint. The client test needs the server to be available to accept requests for the duration of all it's tests. The server, since it follows the conventional JUnit model, wants to set up, test, and then tear down. When the server and then the client tests are run, the `testClientCall()` method of `SimpleClientTest` will pass. However, the `testRequestCount()` will never end and the `tearDown()` method will never be called.

To stop the `SimpleServerTest` from running, the process will have to be killed. If the code is changed so the `testRequestCount()` method of `SimpleServerTest` does not loop, it is very possible that the test run of the `SimpleServerTest` would be finished before the `SimpleClientTest` would have a chance to call on it, thus causing the `testClientCall()` to fail.

decorated

Although this sample does not use the jrunit features directly, it does show the power of using the benchmark decorator. This sample uses a slight variation of the basic `SimpleClientTest` and `SimpleServerTest`.

```
   public static Test suite()
   {
      return new ThreadLocalDecorator(SimpleClientTest.class, 3, 10, 0, true);
   }

   public void testClientCall() throws Exception
   {
      // start benchmark tracking
      ThreadLocalBenchmark.openBench("ClientCall");
      ThreadLocalBenchmark.openBench("GetSocket");

      getSocket();

      ThreadLocalBenchmark.closeBench("GetSocket");

      /**
       * Uncomment this to see that is the same object, thread, and socket for each loop,
       * but will be different for each thread as specified to the test decorator.  This
       * is based on the values passed to JunitThreadDecorator (or one of it's subclasses)
       * for the numberOfThreads and loop parameter values.
       */
      //System.out.println(hashCode() + " - " + Thread.currentThread().getName() + " - " + socket.hashCod

      oos.writeObject("This is the request from " + Thread.currentThread().getName());
      oos.reset();
      oos.writeObject(Boolean.TRUE);
      oos.flush();
      oos.reset();

      Object obj = objInputStream.readObject();
      objInputStream.readObject();

      assertEquals("This is response.", obj);

      ThreadLocalBenchmark.closeBench("ClientCall");
```

```
   }
```

The `SimpleClientTest` was modified to declare a `suite()` method. This method will return a new instance of the `org.jboss.jrunit.decorators.ThreadLocalDecorator` class. In this particular case, the parameters specify that the decorator should run the test case `org.jboss.jrunit.sample.decorated.SimpleClientTest`, using 3 separate threads, each looping 10 times, with no delay (see ThreadLocalDecorator section for more information on configuring the ThreadLocalDecorator).

This means that when the `SimpleClientTest` is run, there will be three instances created that run all the tests 10 times each. The `testClientCall()` method of the `SimpleClientTest` class also has some lines added to mark the starting and ending of named benchmarks. In particular, the `ClientCall` and the `GotSocket` benchmark will be started at the beginning of the method.

The `GotSocket` benchmark will be stopped after making the call the `getSocket()` method and the `ClientCall` benchmark will be closed at the end of the method. At this point the `SimpleServerTest` has not modified. Therefore, based on the code, it will accept only one client socket connection at a time and process the requests for this connection. After that connection is closed, it will move onto the next connection.

To run this sample, start the `org.jboss.jrunit.sample.basic.decorated.SimpleServerTest` and then the `org.jboss.jrunit.sample.decorated.SimpleClientTest`. After the client has finished, stop the server and at the bottom of the client console, should see something similar to:

```
SubBenchmarks: Benchmark:ClientCall Executions:30 Time:15311 SubBenchmarks: Benchmark:GetSocket Executions:30 Time:15191
```

This indicates how long it took for the three threads to run ten iterations of the `testClientCall()` and `getSocket()` methods. Now, uncomment the for loop in the `setUp()` method of the `SimpleServerTest` class. This will make it so there are three threads available to accept incoming client socket request (please do not send me any complaints about how this is NOT how servers should be coded... it for demonstration purposes, relax :)). Compile and re-run. You results should now look something like:

```
SubBenchmarks: Benchmark:ClientCall Executions:30 Time:130 SubBenchmarks: Benchmark:GetSocket Executions:30 Time:30
```

These results indicate that just by changing the server to be multi threaded so that it could accept and process all the client requests concurrently the client calls executed over 100 times faster.

clientserver

The next example, which is under `org.jboss.jrunit.sample.clientserver` package addresses how jrunit helps to solve these issues when running a client/server test within a JUnit construct. The only change required to the original test classes is that the `SimpleServerTest` extends `org.jboss.jrunit.ServerTestCase` instead of `junit.framework.TestCase`, which will allow the behavior change needed within the server test run (see the section on ServerTestCase for more details on how this class changes the test run behavior).

```
public class SimpleServerTest extends ServerTestCase  // NOTE: Not TestCase, but ServerTestCase
{
   ...
```

Also changed the test to use log4j instead of System.out.println for logging since output will not be to console, more on why this is in a minute. The other class within the `org.jboss.jrunit.sample.clientserver` package is

the `SampleClientServerTest`. This class extends the `org.jboss.jrunit.harness.TestDriver` class which provides the harness for running the client and server in different processes.

```
public class SampleClientServerTest extends TestDriver
{
   public void declareTestClasses()
   {
      addTestClasses("org.jboss.jrunit.sample.basic.SimpleClientTest",
                     1,
                     "org.jboss.jrunit.sample.clientserver.SimpleServerTest");
   }

   protected Level getTestHarnessLogLevel()
   {
      return Level.DEBUG;
   }
}
```

The main method of note is the `declareTestClasses()`, which must be implemented as is an abstract method within `TestDriver`. This method then calls the `addTestClasses()` method from the `TestDriver`. The parameters to the `addTestClasses()` method, in order, are the client test class to run, the number of clients processes to spawn, and the server test class to run. Notice that the client specified is the original `org.jboss.jrunit.sample.basic.SimpleClientTest` class and the new `org.jboss.jrunit.sample.clientserver.SimpleServerTest`, which extends `ServerTestCase`.

There is also the `getTestHarnessLogLevel()` method, which tells the driver what the log level should be set to for the test harness code, for debugging purposes. To run this sample, go to the root directory and run the ant target `run-SampleClientServerTest`. The console output should be similar to:

```
run-SampleClientServerTest:
[junit] Running org.jboss.jrunit.sample.clientserver.SampleClientServerTest
[junit] Tests run: 1, Failures: 2, Errors: 0, Time elapsed: 10.045 sec
[junit] Test org.jboss.jrunit.sample.clientserver.SampleClientServerTest FAILED
[junitreport] Transform time: 471ms
```

This ant task will also create an html report under the `output/test-report/` directory. Reviewing the results should indicate that there were two failures. The first being the failure from the `testFailure()` method of the `org.jboss.jrunit.sample.basic.SimpleClientTest` class and the other being from the `testRequestCount()` method of the `org.jboss.jrunit.sample.clientserver.SimpleServerTest` (because only received one client call instead of 30).

multipleclientserver

This next sample is basically the as the previous sample except will run three client processes to call on the server instead just the one. The only new class for this example is the `org.jboss.jrunit.sample.multipleclientserver.SampleMultipleClientServerTest`.

```
public class SampleMultipleClientServerTest extends TestDriver
{
   public void declareTestClasses()
   {
      addTestClasses("org.jboss.jrunit.sample.basic.SimpleClientTest",
                     3,
                     "org.jboss.jrunit.sample.clientserver.SimpleServerTest");
   }
}
```

The only difference in this class and the `SampleClientServerTest` from before is that change the second parameter to `addTestClassses()` method from 1 to 3. To run this sample, go to the root directory and run the ant target `run-SampleMultipleClientServerTest`. The console output should be similar to:

```
run-SampleMultipleClientServerTest:
[junit] Running org.jboss.jrunit.sample.multipleclientserver.SampleMultipleClientServerTest
[junit] Tests run: 1, Failures: 4, Errors: 0, Time elapsed: 11.577 sec
[junit] Test org.jboss.jrunit.sample.multipleclientserver.SampleMultipleClientServerTest
FAILED [junitreport] Transform time: 511ms
```

This ant task will also create an html report under the `output/test-report/` directory. Reviewing the results should indicate that there were now four failures. The first three being the failure from the `testFailure()` method of the `org.jboss.jrunit.sample.basic.SimpleClientTest` class and the fourth being from the `testRequest-Count()` method of the `org.jboss.jrunit.sample.clientserver.SimpleServerTest` (because only received one client call instead of 30). This also means that each of the three `SimpleClientTest`'s `testClientCall()` method executed successfully.

clientonly

It is also possible to just use the jrunit framework for running the client side only in the case where a server is already running. The way to accomplish this, as shown in `org.jboss.jrunit.sample.clientonly.SampleClientOnlyTest` is to change the last parameter to the `addTest-Classes()` method to be `null`.

```
public class SampleClientOnlyTest extends TestDriver
{
   public void declareTestClasses()
   {
      addTestClasses("org.jboss.jrunit.sample.basic.SimpleClientTest",
                     3,
                     null);
   }
}
```

To test this, run the `org.jboss.jrunit.sample.basic.SimpleServerTest`, then the `org.jboss.jrunit.sample.clientonly.SampleClientOnlyTest`. Remember that the `SimpleServerTest` will have to be shutdown manually in this case. The results should be the same as the previous example, except without the failure from the `SimpleServerTest`.

decoratedclientserver

This sample is exactly like the previous client/server examples from before, except the client is `org.jboss.jrunit.sample.decoratedclientserver.SimpleDecoratedClientTest`. This class has the same code as the `SimpleClientTest` in the previous examples, but has added code for obtaining benchmark results in addition to the basic JUnit test results.

```
...

   public static Test suite()
   {
      return new ThreadLocalDecorator(SimpleDecoratedClientTest.class, 10);
   }
```

```
   public void testClientCall() throws Exception
   {
      // start benchmark tracking
      ThreadLocalBenchmark.openBench("ClientCall");
      ThreadLocalBenchmark.openBench("GetSocket");

      getSocket();

      ThreadLocalBenchmark.closeBench("GetSocket");

      /**
       * Uncomment this to see that is the same object, thread, and socket for each loop,
       * but will be different for each thread as specified to the test decorator.  This
       * is based on the values passed to JunitThreadDecorator (or one of it's subclasses)
       * for the numberOfThreads and loop parameter values.
       */
      //System.out.println(hashCode() + " - " + Thread.currentThread().getName() + " - " + socket.hashCod

      oos.writeObject("This is the request from " + Thread.currentThread().getName());
      oos.reset();
      oos.writeObject(Boolean.TRUE);
      oos.flush();
      oos.reset();

      Object obj = objInputStream.readObject();
      objInputStream.readObject();

      assertEquals("This is response.", obj);

      ThreadLocalBenchmark.closeBench("ClientCall");

   }
```

The main differences in code is a new `suite()` method has been added which creates a new `ThreadLocalDecorator`, specifing that client should loop running it's test 10 times. There are also lines within the `testClientCall()` which specify when particular benchmark metrics, via static calls to `ThreadLocalBenchmark`, should be started and stopped. For more details on ThreadLocalBenchmark, please refer to the section on ThreadLocalBenchmark. The `SampleDecoratedClientServerTest` class extends `BenchmarkTestDriver` instead of `TestDriver`. This sets up the root test driver so that it will listen and report the remote benchmark results to the console and to a local file based on the test's class name (i.e. SampleDecoratedClientServerTest_benchmark.txt).

4

Configuration

The TestDriver provides many methods that can be overridden to change the default configuration. These are as follows:

```
/**
 * How long to wait for test results to be returned from the client(s). If goes longer than the
 * specified limit, will throw an exception and kill the running test cases. Default value is * RESULTS
 * @return
 */
protected long getResultsTimeout()
{
   return RESULTS_TIMEOUT;
}

/**
 * How long for the server test case to wait for tear down message. If exceeds timeout,
 * will throw exception. The default value is TEARDOWN_TIMEOUT.
 * @return
 */
protected long getTearDownTimeout()
{
    return TEARDOWN_TIMEOUT;
}

/**
 * How long to allow each of the test cases to run their tests. If exceeds this timeout
 * will throw exception and kill tests. The default value is RUN_TEST_TIMEOUT.
 * @return
 */
protected long getRunTestTimeout()
{
   return RUN_TEST_TIMEOUT;
}

/**
 * Returns the classpath to be added to the classpath used to start the client tests.
 * Default return is null, which means no extra classpath will be added.
 * @return
 */
protected String getExtendedClientClasspath()
{
   return null;
}

/**
 * Returns the classpath to be added to the classpath used to start the client tests.
 * Default return is null, which means no extra classpath will be added.
 * @return
 */
protected String getExtendedServerClasspath()
{
   return null;
}
```

```
   /**
    * Returns the VM arguments to be passed to the vm when creating the client test cases (actually their
    * The default value is null.
    * @return
    */
    protected String getClientJVMArguments()
    {
        return null;
    }

   /**
    * Returns the VM arguments to be passed to the vm when creating the server test cases (actually their
    * The default value is null.
    * @return
    */
    protected String getServerJVMArguments()
    {
      return null;
    }
```

Gotchas

If see something weird like:

```
junit.framework.AssertionFailedError: Method "Yourclass.class" not found
```

check to make sure you do not have a constructor in your test class that looks like:

```
public Yourclass(String name) { super(NAME); // where NAME = "Yourclass.class" }
```