Capabilities of a Java Test Execution Framework *by Erick Griffin*

Here we discuss key properties and capabilities of a Java Test Execution Framework for writing and executing discrete Java tests for testing Application Programming Interfaces (APIs)/methods or GUIs in Java. Tivoli's Java Execution Framework (JEF) is a multi-purpose Java test execution framework that, when used with other tools or methodologies, can provide a framework for testing Java applications. Moreover, since this framework contains many open and ease-of-use features it can be used in all phases of test, providing a consistent and flexible test infrastructure throughout the verification organization.

Prior to the launch of any major product development cycle Software Engineers responsible for the testing of a new product have to ask themselves one simple question, "Do we have the tools available to test this new product?" We were faced with just such a question late in 1998 when our company planned an ambitious development effort. This included a cadre of new products and applications written in Java. Fortunately, our Test Automation Group (which I am a member of) had enough lead time to analyze and plan a solution for this problem. This naturally involved looking at all of the existent tools that were available at the time. However, since I was responsible for this analysis, I quickly came to the conclusion early in 1999 that due to special requirements within our corporation and test infrastructure, we had to design and develop our own Java Test Tool solution.

A Problem of Simplicity

One of the most fundamental problems to overcome when designing a tool is simplicity. Not only did our tool have to be easy enough to use for first time users, but it also had to meet the needs of more advanced users. I am by my very nature a minimalist, so early in the design phase our design group quickly identified that a main component of any tool that we created had to be simple and easy to use. Additionally, it had to quickly fit into whatever the existing build and test structure was. It would have to have a Command Line Interface (CLI) and provide many of the features that current Unix CLI-based programs and applications possessed. Other general requirements were:

- A simple set of Application Programming Interfaces (API) or Methods for writing **Testsuites** and **Testcases**.
- An advanced set of APIs, Methods and Interfaces to extend **Testsuites** and **Testcases** for testing more complex problems.
- A general use Test Harness that would provide automation for collecting and driving Testsuites and Testcases for any test phase.

Essential Ingredients of a Java Execution Environment:

- Simple Testsuite / Testcase definition.
- Interfaces to control execution.
- A Harness to introspect and drive test execution through all phases of test.
- A Data Base mechanism to store automated test results.

After deliberating over the tool capabilities it was clear that this would be an execution framework or environment and the name ***JEF*** was quickly adopted, an acronym for the ***Java Execution Framework***.

What is a Testsuite? What is a Testcase?

A definition of what constituted a **Testsuite** and a **Testcase** would need to be simplistic as well. A **Testsuite** could contain two methods, one for initialization ***prepSuite***, and one for cleanup ***postSuite***. The initialization method could return an array of Objects that in turn would be passed into each **Testcase** within that **Testsuite**. Though optional, using such a class means that a certain level of test initialization can be performed in one place and cleaned up. This in turn makes each **Testcase** simpler and easier to understand since duplication of essential but common tasks is removed from each. Each **Testcase** on the other hand can also optionally contain its own **Testcase** initialization and cleanup methods, ***startTest*** and

endTest respectively. Within the formal definition of a **Testcase** however, a user is required to provide one method, a ***runTest*** method, to perform essential testing.

Writing a simple Testcase is therefore easy for all it needs to do is to extend the ***runTest*** method of the abstract class ***JEFTestCase***. Again, using any of the other methods mentioned above is purely optional. Once this is properly conveyed to the users, they are eager to embrace the API set and test framework. A sample of a simple **Testcase** written in the JEF API set is provided in Figure 1.

```
//--------------------------------------------------------------------
package com.tivoli.usersGuide.samples_PartI;
import com.tivoli.jefapi.*;
import java.io.*;
import java.util.*;
import java.lang.*;

public class JEFSimpleTest0 extends JEFTestCase {
  //
  // A main so this Testcase can be called from the command line as described
  // in the documentation.  However, when run with the JEF Test Harness, the
  // runTest method is called directly, so no test depedent code is placed here.
  //
   public static void main( String[] args ) {
      JEFSimpleTest0 st1 = new JEFSimpleTest0();
      try {
         st1.startTest();
         st1.runTest( args, null );
         st1.endTest();
      } catch ( Exception e ) {}
   }

  /**
   * A JEFTestCase that just says "Hello World...!!".  This Testcase has NO
   * <em>startTest</em> nor <em>endTest</em> methods, these methods are
   * to be picked up from the super class.
   */
  public void runTest( String args[], Object obj[] ) throws Exception {
      // Send a HELLO WORLD
       jefUtil.log( "Hello World...!!" );
  } // end method
} // end JEFSimpleTest0 class
```

Figure 1.
A Sample Testcase, the traditional Hello World test.

However, many testers still had fundamental questions regarding what constituted a **Testcase**. My response to them was always clear. A **Testcase** should test one discrete function of an API or CLI. Unfortunately, you cannot always take a purely academic approach to such matters and I quickly introduced the concept of **Testcase variations**. These are actually just sub-function tests within a larger test. With their results independently tracked, many users opted to use them in order to avoid the explosion of **Testcase** writing that could ensue when many functions within a component have the same initialization and test attributes. Again, a simple example of this is given in Figure 2. This figure also shows two ways to set the **Testcase variation's results**, the first one using the *setVarResult* method and the latter using the *endVar* method directly. Additionally, variation results are always taken into account when setting overall **Testcase** results, however, the user can always override this using the *setResult* method for the **Testcase.**

The above figures and examples provide only a brief overview of the rich API set that JEF possesses. For more detail on the entire functionality of JEF, the reader is directed to the ***JEF User's Guide***.

```
/**
  */
  public void runTest( String args[], Object obj[] ) throws Exception {
    // Send a HELLO WORLD for two variations
     myVar1();
     myVar2();
  } // end method

/**
  */
  private void myVar1() {
     startVar("myVar1");
     jefUtil.log( "Hello World from variation 1 !!" );
     setVarResult(jefState.jefStatus.PASSED);
     endVar();
  }

/**
  */
  private void myVar2() {
     startVar("myVar2");
     jefUtil.log( "Hello World from variation 2 !!" );
     endVar(jefState.jefStatus.PASSED);
  }
```

Figure 2.
Using Testcase Variations.

A Question of Control

Of course the next question the user must ask themselves is how do we control test execution with regards to logging information, tracing important problem determination data, etc. The **JEF** API set also incorporates all the features that one should expect in this area. In both cases a utility class is provided that provides threaded log and trace support. The user gets this class for free when they extend the abstract **Testcase** class. To use it, they simply call one of the forms of *jefUtil.log()* or *jefUtil.trace()*. Both of these methods have forms that can log or trace objects, methods and any parameters they may take. Figure 2 shows one example of how the log method could be used for simple informational type messages.

JEF also has state and status classes. These are used to keep track of Testcase information and the status of the execution results. The classes *JEFState* and *JEFStatus* respectively, are automatically provided as part of instantiating a Testcase that extends *JEFTestCase*. The status class *JEFStatus* is actually part of the state class *JEFState*, so if the user were to get their own state object they would get the status object as well.

The state object contains the Testcase name, Testcase class name, the Testsuite name that it belongs to (if any) and its class, a product name, a product release, a component name, an organization name, a build designation, a driver specification and test phase specification. The status object on the other hand contains an object to hold the results of the Testcase and any Testcase variations. Moreover, it also contains variables to hold any Testcase directives that might be set by the Testcase.

Another initial requirement was the ability to stop the execution of a Testcase in an orderly manner. A **Testcase directive** is a Testcase command by the Testcase to the **JEF Harness** instructing it to halt execution. This is either to end execution for an entire test run or just for a particular Java Testcase package name. This is very useful when executing large numbers of Testcases and an important resource or service is unavailable or unexpectedly fails. In most cases it no longer makes sense to continue Testcase execution for the entire test run or the affected package. A directive therefore gives a Testcase writer the ability to halt execution under certain conditions, in order to meet the requirements of a particular test.

```
/**
  * Our Testcase class.
  */
public class JEFSimpleIJEFTimeOut extends JEFTestCase
                                  implements IJEFTimeOut {

   public static long ijefTimeOut = 3000; // timeout in milliseconds.

   public void runTest( String args[], Object obj[] ) throws Exception {

     // This just loops and checks to see if our Testcase is interrupted
     // during execution.
      while( !isInterrupted() ) {
        // Do your important work and calls here...
         if ( isInterrupted() ) break;
      } // endwhile
   } // end method

   public boolean isInterrupted() {
      if ( java.lang.Thread.currentThread().isInterrupted() ) {
         return( true );
      } else {
         return( false );
      } // endif
   } // end method
}
```

There are also some important interfaces the user can use to control the execution of their Testcase. One such interface is the *IJEFTestArgument interface* where the Testcase writer can specify a number of Testcase arguments for separate Testcase invocations. That is, this interface enables the user to define static arguments for separate invocations of the same Testcase. When used in conjunction with the **JEF Harness,** a Testcase like the one shown in Figure 3., would be invoked four different times, each time with a new set of arguments as defined on the interface. This useful interface can be used by Testcase developers to ensure that a default set of arguments (e.g. boundary conditions, etc.,) will always be used when arguments are not supplied by user via other means.

Another useful interface is the *IJEFTimeOut interface*. As the example in Figure 4 shows, this interface enables the Testcase writer to define a timeout value in milliseconds for the maximum amount of time the Testcase can execute. When this value is exceeded the Testcase is sent an interrupt. Since all Testcases run on their own thread, it is expected to catch that interrupt by implementing and using the *isInterrupted* method. This type of thread model avoids the use of deprecated Java Thread APIs and methods that are inherently unsafe. Therefore it is incumbent on the user to conform to the interface in order for it to work

```
public class JEFSimpleIJEFTestArguments extends JEFTestCase
                                        implements IJEFTestArguments {

  // Our arguments for test.
   public static String testArguments[][] = { {"Hello ","World! ","- in English"},
                                              {"Hallo ","Welt! ","- in German "},
                                             {"Ciao ","mondo! ","- in Italian "},
                                              {"Allo ","monde! ","- in French "}
                                            };

   public void runTest( String arg[], Object obj[] ) throws Exception {
     // test body. . .
   } // end method

} // end class
```

Figure 3.
Defining multiple Test Arguments using the *IJEFTestArguments* interface.

correctly. Moreover, from a practical point of view this allows the Testcase to perform the appropriate functions, such as returning any resources that it may have been using at the time, prior to returning.

A Harness to Drive Execution

Sine **JEF** is an execution framework it also contains a test harness to drive execution. The harness provides many features to help users group and control **Testsuites** and **Testcases** and their execution.

To begin with, the **JEF Harness** provides the capability to invoke **Testsuites** and **Testcases** independent of classpath. There is one requirement on the classpath, it must contain the necessary path to the jar-file for **JEF** and the current working directory designation (e.g. '.') and that is all. This has the effect of removing from the user the need to worry about the classpath designation during test execution, whether the **Testcase** belongs to a package or not.

Of course the **JEF Harness** possesses a rich Command Line Interface (CLI) for users to perform many different operations. A list of these capabilities would consist of the following:

- Help option, the user can always obtain help regarding the usage of each option of parameter.
- Auto-Generation option, the user can auto-generate important files for the control of execution.
  - A properties file that contains properties that can be changed by the user to control execution.
  - SuiteFiles (e.g. mytest.suite) in flat file format, which contain the contents of a particular test run. This file contains the classpath used at the time and any **Testsuites** and their **Testcases** that were selected based upon other options and parameters.
- Execute option, to instruct the **JEF Harness** to perform test execution for this test run.
- Prompt option, to place the **JEF Harness** into user interactive mode during execution.
- Report option, to instruct the **JEF Harness** to report test results to a Test Results Repository database.
- Version option, so the user can see what version of the **JEF Harness** they are using or have installed.
- High-Level Qualifier parameter, indicating what high-level qualifier the user wants to use when the property files are read.
- Property File parameter, a designated property file to use, otherwise *configurableProperties.txt* is assumed and used.
- Keyword Expression parameter, the user can specify a simple regular expression including **keywords** that **Testcases** have defined. This enables the user to select only those **Testcases** for a particular test run. A **Testcase** can have more than one **keyword** defined, depending of course on what functional components the execution of this **Testcase** may affect. An expression can contain logical AND, logical OR, NOT and parenthesis.
- SuiteFile parameter, a user specified suite file that was previously generated by the user.
- Testsuite parameter, a user specified **Testsuite** to execute.
- Testcase Name parameter, a user specified **Testcase name**. If no **Testcase class** is specified, this could either be a simple name of the **Testcase** in a **suiteFile** or it could be the name of the class file to execute.
- Testcase Class parameter, a user specified **Testcase class**. This is the actual class name of the **Testcase** to use. If the user uses the Testcase Name parameter too they can assign a simple name to this class to simplify executing it later.
- Testcase Results File parameter, a user specified file name to contain the results of a test run. This can be used later with the report option to populate the Testcase Result Repository database.

Although the above list does not completely reflect the environments total capabilities, the reader should be able to get some sense of its richness. With the aforementioned, the **JEF Harness** can provide support for executing a simple **Testcase** in the most mundane cases to assembling and executing many **Testsuites** based on keywords and other parameters. Therefore a great deal of flexibility can be obtained when using the harness during execution, although this is not necessarily a requirement.

A Method of Storing Results

A **Testcase** must also be able to store its results somewhere. In some cases a simple log file is sufficient and the **JEF API** set provides this base level of functionality with its utility class. However, there comes a time when overall test statistics need to be rolled up and reported to a common source or database. In today's test environments this is an ever-present requirement. Not only will this be used by management to track overall status of a test effort, but if done correctly it can also be used by the verification staff themselves to obtain real time infomation on test execution.

In the latter stage of the design and development of **JEF** one of the key issues were where should the tool provide test results. Many might pass this off as an obvious concern, but when looked at further it is not always obvious or intuitive where a verification engineer ought to expect their test output to appear. Luckily, our group had other examples that were already in place that I could use for reference. A final list of requirements was soon decided upon which included the following.

- A *testcase_name.log* file for storing information and data as logged by Testcase during execution. Again, the utility class as discussed earlier provides this.
- A *testcase_name.trace* file for storing diagnostic data when exceptions occurred and where traced by the Testcase. Again part of the utility class discussed earlier.
- A logging/tracing level mechanism to filter what types of logging and tracing messages were allowed to appear in either the log or trace files respectively (e.g. INFORMATIONAL, WARNING, ERROR, etc…).
- A local file to store results in the Testcase repository database. Since it wasn't always desirable to report the results immediately over a network connection, a local copy of what would be reported is created in the current working directory. This can later be recorded by using the appropriate options and parameters on a subsequent call to the harness.
- A centralized database for collecting overall test results when reported.

Of course the reporting function is rarely used during unit and functional verification, since these test phases are more interactive and normally require constant surveillance and attention. However, depending upon the test process and plans used this would most certainly be done at least once at the end of each of these test phases. Other test phases that follow are more fortunate since in one respect tests developed for unit and functional verification are only run as a regression test. In this case the system tester doesn't normally expect constant failures and can therefore use some additional functionality supplied by a database-monitoring tool that our group provides. This tool allows the tester to keep track of multiple test runs simultaneously by querying the Testcase results repository database and reporting that data in a Java application as it changes in real time. Therefore, if a system tester were using **JEF** to control their test runs and used the report option, then they would be able to watch multiple tests runs from one central location without babysitting each individually in the laboratory.

A Last Word

Though there were many attempts by us to find and identify a third party tool that would fulfill our requirements in the Java Testing Framework area, we had to reconcile ourselves to designing and developing our own. Once this decision was made we ensured that we built into it a simple **Testsuite** and **Testcase** definition, essential interfaces to control execution, a harness to enhance and exploit test execution characteristic that were built into the framework and a Testcase results repository to capture test execution results. Now, after more than a year and a half of use both its simplicity and overall capabilities have paid off well with wide spread acceptance and use throughout our company. So much so, that we continue to enhance its capabilities and integration with other components and tools of our company's overall test automation approach. We even use **JEF** to test **JEF** itself.