

Chapter 4

JSystem Framework Services



➤ In this chapter...

<i>JSystem Framework Services Overview</i>	<i>Page 2</i>
<i>Framework Services Interaction with Automation Layers</i>	<i>Page 3</i>
<i>How to Write a System Object</i>	<i>Page 4</i>
<i>System Under Test (SUT) Independence</i>	<i>Page 8</i>
<i>Reports</i>	<i>Page 17</i>
<i>HTML Reporter</i>	<i>Page 21</i>
<i>Analyzers</i>	<i>Page 34</i>
<i>Monitors</i>	<i>Page 38</i>
<i>Fixtures</i>	<i>Page 39</i>
<i>Multi-User Support</i>	<i>Page 48</i>
<i>Additional Services</i>	<i>Page 72</i>
<i>Additional Language Support</i>	<i>Page 75</i>

4.1 JSystem Framework Services Overview

The JSystem Framework services refer to the set of API software applications provided by the JSystem Automation Platform that enable the user to develop and provide improved and streamlined project automation.

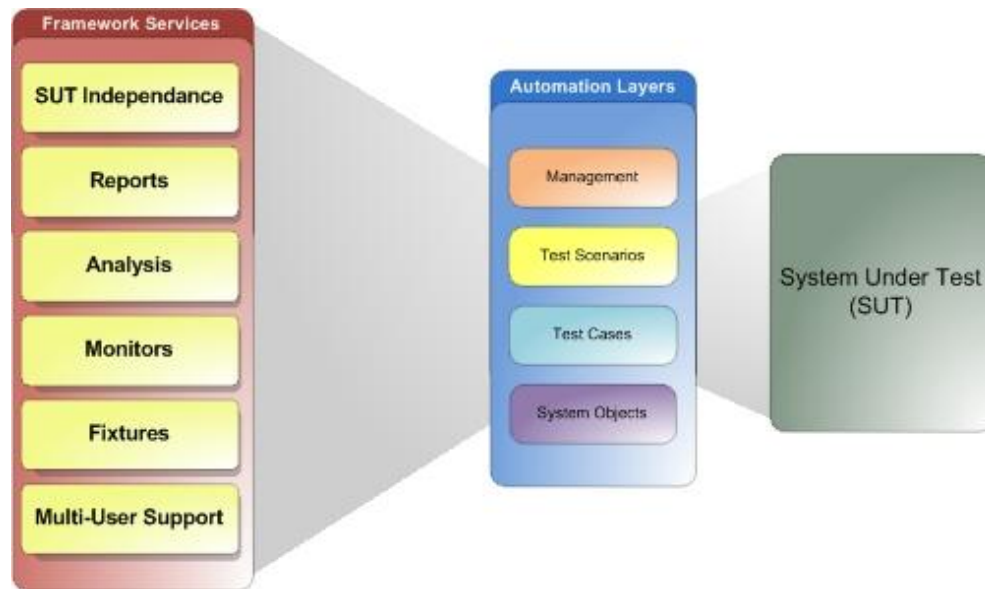


Figure 1: JSystem Services Illustration

The drawing illustrates the JSystem services interaction with the automation layers and the SUT. The framework service layers are the building blocks of the JSystem Automation Platform.

The following section explains in detail the Framework Services using Java code examples to show how these building blocks are constructed and used.

4.1.1 Framework Services Interaction with Automation Layers

SUT Independence - The SUT Independent Framework Service interacts directly with the system object. This service independence refers to the ability of the user to apply the same test to different DUT and SUT products and change parameters within the tests.

Reports - The Reports Framework Service interacts directly with the system object layer and the test/fixtures layer, providing transparent information access to all other automation layers. When a scenario is run by JSystem the test case layer operates the system objects on the SUT or DUT the report framework service then extracts the results. These results are delivered to the reports framework service and are collated in the central management report mechanism called the **JReporter** application.

Analysis - The Analysis Framework Service extracts statistics from the Report Framework Service connecting directly to results produced by the Test/Fixture layer. The Test/Fixture layer sends a request to the System Object layer for an analysis of a specific function; the results are then collated via the Report Framework Service and sent to the user.

Monitor - The Monitor Framework Service runs a service that is performed in parallel to the test being performed. The Monitor Framework Services are written by the onsite Java programmer, according to the specification requirements of the product being tested.

Fixture - The Fixture Framework Service connects directly to the Test/Fixture and the Scenario layers, similar the tests the fixtures are written by the onsite Java programmer. The objective of the fixture is to bring the DUT to a state that enables the JSystem to perform tests. A fixture can be assigned to either a scenario or a Test/Fixture layer.

System Object Lifecycle - The system object lifecycle service controls the initiation, system object state during execution and termination of the system object, and helps the user to implement a system object that works well with the pause and graceful stop features de-allocating resources when the execution ends.

Multi User Support - The Multi User Framework Service provides the test implementer a set of test variables that can be applied to a test or fixture via the scenario loaded the JSystem JRunner. This provides extended functionality and capabilities allowing the user to create and customize tests within the scenario by changing value options from within the **"Test Info"** tab. The user can construct variations to the test that sits inside the scenario. These selections are dynamic and intelligent, meaning that both the tasks and their content change dynamically according to the variable values chosen from the **"Test Info"** **"Sub-Tab"**.

4.2 System Objects

JSystem system objects (also called drivers) are java objects which encapsulates the code that interacts with the SUT. System objects benefit from JSystem services as described later on in this chapter.

4.2.1 How to Write a System Object

The Following guidelines provide the necessary steps required to write and implement system objects.

1. The System object class should extend the class `"jsystem.framework.system.SystemObjectImpl"`
2. The **"system object"** must have a default constructor.
3. The system object class members should be exposed with setters and getters.
4. The system object initialization should be implemented in the **"public void init() throws Exception"** method. If there is no need for specific system object initialization, do not implement the method. If it is required then call the **"super.init()"** method.
5. The system Object resource release and finalization should be implemented in the **"public void close()"** method. If there is no need to finalize a specific system object, do not implement the method. If it is required call the **"super.close()"** method.
6. All system object operations should be exposed as plain public java methods.
7. As with java objects, system objects can be initialized directly by developer code, in order to enjoy the SUT independence and lifecycle management services let the system instantiate the system object by calling the **system.getSystemObject("name")** method.

Note: The purpose of the system object is to hide the complexity of SUT interaction, enabling test authors the ability to avoid dealing with these complexities. In order to stream line testing operations only expose simple and well documented methods to the user.

4.2.1.1 System Object Example

```
01 package com.aqua.services.demo;
02 import jsystem.framework.system.SystemObjectImpl;
03 import com.aqua.sysobj.conn.CliCommand;
04 import com.aqua.sysobj.conn.CliConnectionImpl;
05 import com.aqua.sysobj.conn.WindowsDefaultCliConnection;
06 public class SimpleWindowsStation extends SystemObjectImpl {
07     private String host;
08     private String userName;
09     private String password;
10     private CliConnectionImpl cliConnection;
11     public void init() throws Exception {
12         super.init();
13         cliConnection = new WindowsDefaultCliConnection(getHost(), getUserName(), getPassword());
14         cliConnection.init();
15         report.step("In init method");
16     }
17     public void close() {
18         report.step("In close method");
19         cliConnection.close();
20         super.close();
21     }
22     public void mkdir(String folderName) throws Exception {
23         CliCommand cmd = new CliCommand("mkdir " + folderName);
24         cmd.addErrors("unknown command");
25         cliConnection.handleCliCommand("created dir " + folderName, cmd);
26         setTestAgainstObject(cliConnection.getTestAgainstObject());
27     }
28     public void dir(String folderName) throws Exception {
29         CliCommand cmd = new CliCommand("dir " + folderName);
30         cmd.addErrors("unknown command");
31         cliConnection.handleCliCommand("dir " + folderName, cmd);
32         setTestAgainstObject(cmd.getResult());
33     }
34     public void ping(String host) throws Exception {
35         CliCommand cmd = new CliCommand("ping " + host);
36         cmd.addErrors("unknown command");
37         cliConnection.handleCliCommand("ping " + host, cmd);
38         setTestAgainstObject(cliConnection.getTestAgainstObject());
39     }
40     public String getHost() {
41         return host;
42     }
43     public void setHost(String host) {
44         this.host = host;
45     }
46     public String getUserName() {
47         return userName;
48     }
49     public void setUserName(String userName) {
50         this.userName = userName;
51     }
52     public String getPassword() {
```

```

53     return password;
54 }
55 public void setPassword(String password) {
56     this.password = password;
57 }
58 }

```

Figure 2: System Object Code Example

4.3 JSystem Tests

JSystem tests are simple JUnit tests that can be extended in order to support Jsystem multi-type user service.

Traditionally, jsystem tests are Junit 3.x tests, since JSystem 5.1 we support JUnit4 tests. All Jsystem services support Junit3 and JUnit4.

New tests should be written in JUnit4.

4.3.1 JUnit 4 Support for Test Authoring

JUnit 4 was originally released two years ago, it brought about a new cleaner style of writing test cases. Tests written in the JUnit 4 style no longer depend on a specific method naming convention (starting each method name with the word *test*, calling the set up and tear down methods *setUp* and *tearDown*). Instead, the user can use annotations to mark methods as tests as well as additional methods to be run before or after test methods.

This latest version of JSystem adds s JUnit 4 support. JSystem still supports the Junit 3 test style.

```

01 public class MyTestCase extends SystemTestCase {
02     public MyTestCase() {
03         super();
04         setParentFixture(BasicFixture.class);
05     }
06     public void setUp() {
07         report.step("MyTestCase setUp");
08     }
09     public void tearDown() {
10         report.step("MyTestCase tearDown");
11     }
12     public void testSomething() {
13         report.step("Testing something");
14     }
15 }

```

Table 1: JSystem test based on JUnit 3 Code Example

4.3.1.1 JUnit4 Example

The user can use the cleaner JUnit 4 style in the following manner.

```
01 public class MyTestCase extends SystemTestCase4 {
02     public MyTestCase() {
03         super();
04         setParentFixture(BasicFixture.class);
05     }
06     @Before
07     public void before() {
08         report.step("MyTestCase setUp");
09     }
10     @After
11     public void after() {
12         report.step("MyTestCase tearDown");
13     }
14     @Test
15     public void something() {
16         report.step("Testing something");
17     }
18 }
```

Table 2: JUnit MyTestCase Code Example

The changes that were made to the code in transition from JUnit 3 to JUnit 4 style are detailed as follows.

- The test case class extends the "**SystemTestCase4**", instead of the "**SystemTestCase**".
- The **@Before** and **@After** annotations mark methods that are run during the test set up and tear down respectively. All the **@Before** methods are called before each test is run and all the **@After** methods are called when it ends.
- Test method names do not need to start with the word **test**. The test author can name the test as required, as long as the test is annotated with the **@Test** annotation.
- Other JUnit 4 annotations:
 - a. **@BeforeClass** – will be executed before the @Before annotation
 - b. **@AfterClass** – will be executed after the @After annotation
 - c. **@Ignore** – test will not be executed

All the old test written in the JUnit 3 style test cases run without any problems. Only tests deriving from "**SystemTestCase4**" will be treated as JUnit 4 style tests.

4.3.2 JUnit 3 support for Test Authoring

Coding guidelines to write JUnit 3 based tests:

1. Extend the class "**junit.framework.TestCase**"
2. Follow JUnit coding guidelines and rules.

Later on in this chapter we discuss how to extend Junit class in order to benefit from jsystem services.

4.4 System Under Test (SUT) Independence

SUT Independence is the ability to run the same test on different setups without changing the test.

Every test setup is comprised of a set of “**SystemObjects**” also known as drivers. One of the framework requirements is that test that was written for one SUT can be executed on a different SUT without changing the test code. All the setup specific information should be placed in the SUT files. The SUT files are XML files that describe the SUT. Every SUT is allocated its own independent SUT file.

4.4.1 What is the SUT

The SUT is the device and or software being tested, as well as they peripheral testing software and hardware that is used to check the DUT. When planning an automation project one of the first steps required is to define the SUT.

Potential elements that can be included in the SUT include:

- Device and or software being tested.
- Workstations or PC's that are part of the setup to be managed.
- A generator of any kind.
- Dynamic configuration.

4.4.2 How the SUT Service Works

The SUT independent service works in the following manner: when the test author writer requires the use of a system object, the state of the system object is loaded from the SUT file using java reflection.

4.4.2.1 SUT Code Example

The following code represents a SystemObject code example.

```
01 package com.aqua. services.systemobject;
02
03 import jsystem.framework.system.SystemObjectImpl;
04 import jsystem.utils.FileUtils;
05
06 public class MyDevice extends SystemObjectImpl{
07
08     private String message;
09     private String fileName;
10
11     public void init() throws Exception {
12         super.init();
13         report.report("Hello world init");
14     }
15     public void close(){
16         report.report("Hello world close");
17         super.close();
18     }
19     public void getHelloMessage() throws Exception {
20         report.report("Hello Message",getMessage(),true);
21     }
22     public void readFromFile() throws Exception {
23         String textFromFile = FileUtils.read(getFileName());
24         setTestAgainstObject(textFromFile);
25     }
26     public String getMessage() {
27         return message;
28     }
29     public void setMessage(String message) {
30         this.message = message;
31     }
32     public String getFileName() {
33         return fileName;
34     }
35     public void setFileName(String fileName) {
36         this.fileName = fileName;
37     }
38 }
```

Table 3: System Object Code Example

4.4.2.2 SUT Test Code Example

This code example shows a test that uses the “**MyDevice**” system object.

```
01 package com.aqua.tascourse;
02 import junit.framework.TestCase;
03 public class ExampleTest extends TestCase {
04     private MyDevice myDevice;
05     public void setUp() throws Exception {
06         myDevice = (MyDevice)system.getSystemObject("helloWorld");
07     }
08     /**
09      */
10     public void testHelloWorld() throws Exception{
11         myDevice.getHelloMessage();
12     }
13     /**
14      */
15     public void testPing() throws Exception {
16         myDevice.ping();
17         // ...
18         // ...
19     }
20 }
```

Table 4: SUT Code Example

4.4.2.3 SUT File Example

The following code example illustrates a simple SUT file.

```
1 <sut>
2     <helloWorld>
3         <class>com.aqua.tascourse.MyDevice</class>
4         <message>Hello TAS Course</message>
5         <fileName>C:/TAS/workspace/java/examples/capture.txt</fil
eName>
6     </helloWorld>
7 </sut>
```

Table 5: SUT Independence Service Example

4.4.3 Behind the Scenes of the SUT Service

The following steps detail the flow of the SUT independent service operations.

1. The test author uses the "**getSystemObject**" method of the system service.
2. The system service refers to the properties file called: "**system.properties**" and searches for the name of the current SUT file.
3. JSystem then loads the SUT file and searches for the entity called "**helloWorld**".
4. Once the file is found the class entity is found and then the java reflection is applied in order to create a new instance of the system object class.
5. The next step in the instancing process of the system object is that the framework revises the entries under the system object entry.
6. Each entry prepends the word "**set**" to the name of the entry and converts the first letter of the entry to a capital letter; this creates a method name the system now searches for this setter method using java introspection, if the method is found it is activated.
7. Once the System Object is initiated and before it is returned from the SUT service, the system object is then entered into an internal map.

4.4.3.1.1 In Our Example

8. The SystemObject class of the helloWorld example is "**com.aqua.tascourse.MyDevice**" after instancing the class, it looks for additional entries.
9. In the "**helloWorld**" system object entry there are two additional entries to the class entry, the message entry and the filename entry.
10. The SUT independent service now searches for the "**setMessage**" setter and setFileName setter.
11. If the "**setMessage**" and "**setFileName**" setters are found they are activated with the vales found in the SUT file "**Hello TAS Course**" and "**C:/TAS/Workspace/java/examples/captures.txt**"

4.4.4 Nested System Object Support

The system object has the ability to contain within itself nested system objects; this enables the system object with the ability to load the configuration of the nested system object from the SUT file.

4.4.4.1 Nested System Object Code Example

The following example of code illustrates the MyDevice system object with a CLI connection member being added to it. The CLI connection is a system object is being used in this example it demonstrate how to initiate a nested system object.

Reference: For further information about CLI connections refer to the Error! Reference source not found. in Chapter 11, on page Error! Bookmark not defined..

```
01 package com.aqua.services.systemobject;
02 import jsystem.framework.system.SystemObjectImpl;
03 import com.aqua.sysobj.conn.CliCommand;
04 import com.aqua.sysobj.conn.CliConnection;
05 public class MyDeviceWithNestedSO extends SystemObjectImpl{
06     private String message;
07     public CliConnection connection;
08     public void init() throws Exception {
09         super.init();
10     }
11     public void close(){
12         super.close();
13     }
14     public void getHelloMessage() throws Exception {
15         report.report("Hello Message",getMessage(),true);
16     }
17     public void dir() throws Exception {
18         CliCommand command = new CliCommand("dir");
19         connection.handleCliCommand("Dir command",command);
20         setTestAgainstObject(command.getResult());
21     }
22     public String getMessage() {
23         return message;
24     }
25     public void setMessage(String message) {
26         this.message = message;
27     }
28 }
```

Table 6: Nested System Object Code Example

4.4.4.2 SUT File Example

The example shows the SUT file of the “**MyDeviceWithNestedSO**” system object. Pay attention to the “**connection**” entity which initiates the nested CLI system object.

***Note:** The name of the class member must be identical to the name of the entry in the SUT file. (As appears in the example, “connection”)*

```
01 <helloWorld>
02   <class>com.aqua.services.systemobject.MyDeviceWithNestedSO</
    class>
03     <connection>
04       <class>com.aqua.sysobj.conn.WindowsDefaultCliConnection
05       </class>
06       <user>simpleuser</user>
07       <password>simpleuser</password>
08       <host>127.0.0.1</host>
09     </connection>
10     <message>Hello TAS Course</message>
11   </helloWorld>
12 </sut></sut>
```

Table 7: Nested SO SUT File Example

4.4.5 Array Support

The system object has the ability to contain within itself a nested array of system objects, this enables the system object with the ability to load the configuration of the nested array of system object from the SUT file.

4.4.5.1 Nested Array of System Object Code Example

The following example shows a “**MyDeviceWithNestedSOArray**” system object with an array of CLI connections.

```
01 package com.aqua.services.systemobject;
02 import jsystem.framework.system.SystemObjectImpl;
03 import com.aqua.sysobj.conn.CliCommand;
04 import com.aqua.sysobj.conn.CliConnection;
05 public class MyDeviceWithNestedSOArray extends SystemObjectImpl{
06     private String message;
07     public CliConnection[] connections;
08     public void init() throws Exception {
09         super.init();
10     }
11     public void close(){
12         super.close();
13     }
14     public void getHelloMessage() throws Exception {
15         report.report("Hello Message",getMessage(),true);
16     }
17     public void dir() throws Exception {
18         CliCommand command = new CliCommand("dir");
19         connections[0].handleCliCommand("dir",command);
20     }
21     public String getMessage() {
22         return message;
23     }
24     public void setMessage(String message) {
25         this.message = message;
26     }
27 }
```

Table 8: Nested Array Code Example

4.4.5.2 SUT File Example

In the example, the SUT file of the “**MyDeviceWithNestedSOArray**” system object that holds an array of connections.

```
01 <sut>
02   <helloWorld>
03     <class>com.aqua.services.systemobject.MyDeviceWithNestedSO
Array</class>
04       <connections index="0">
05         <class>com.aqua.sysobj.conn.WindowsDefaultCliConnection</cl
ass>
06           <user>simpleuser</user>
07           <password>simpleuser</password>
08           <host>127.0.0.1</host>
09       </connections>
10     <connections index="1">
11       <class>com.aqua.sysobj.conn.WindowsDefaultCliConnection</
class>
12         <user>simpleuser</user>
13         <password>simpleuser</password>
14         <host>127.0.0.1</host>
15     </connections>
16     <message>Hello JSystem Course</message>
17   </helloWorld>
```

Table 9: Nested Array SUT File Example

4.4.6 Reference Support

This function enables the user to refer from one element in the SUT file to another element in the SUT file; as a result it avoids unnecessary duplication in the SUT file.

Note: *This function enables the test author to write less without losing functionality.*

4.4.6.1 Reference Support SUT Code Example

The system object is the **"MyDeviceWithNestedSO"**. In order to work with a reference in the SUT file there is no need to make changes to the code. The only changes required are made in the SUT file.

```
01 <helloWorld>
02   <class>com.aqua.services.systemobject.MyDeviceWithNestedSO
</class>
03   <message>Hello TAS Course</message>
04   <fileName>C:/TAS/workspace/java/examples/capture.txt</file
Name>
05   <connection ref="connections/connection[@index='0']">
06     <host edit="enable">10.0.0.93</host>
07   </connection>
08 </helloWorld>
09 <helloWorld2>
10   <class>com.aqua.services.systemobject.MyDeviceWithNestedSO
</class>
11   <connection ref="connections/connection[@index='0']"/>
12 </helloWorld2>
13 <connections>
14   <connection index="0">
15     <class>com.aqua.sysobj.conn.WindowsDefaultCliConnection<
/class>
16     <user>simpleuser</user>
17     <password>simpleuser</password>
18     <host>127.0.0.1</host>
19   </connection>
20   <connection index="1">
21     <class>com.aqua.sysobj.conn.WindowsDefaultCliConnection<
/class>
22     <user>simpleuser</user>
23     <password>simpleuser</password>
24     <host>10.0.0.12</host>
25   </connection>
26 </connections></sut>
```

Table 10: Reference Support Code Example

The SUT file defines two system objects **"hellowWorld"** and **"helloWorld2"**, the definition of the connection entity of the system objects is taken from another section in the SUT file, the **"connections"** section. As can be seen, the connection does not have to be defined twice; it requires to be defined once in the connections section.

4.5 Reports

The JSystem report service provides a broad range of features and functions that enable the user to easily configure the reporting mechanism to the project requirements, they include:

- HTML based reporting mechanism.
- Easily add graphs and diagrams.
- Easily add attachments and links.
- Hierarchical structure.
- Rich report UI (html hierarchy, links, graphs and formatting).
- Pluggable framework.

The Reporting service is an important element required for the visibility of an automation project. The reporting mechanism enables the user to quickly analyze the cause of any test execution failure.

Note: *It is recommended that the user establishes a reporting convention before the commencement of an automation project.*

4.5.1 Reports Architecture

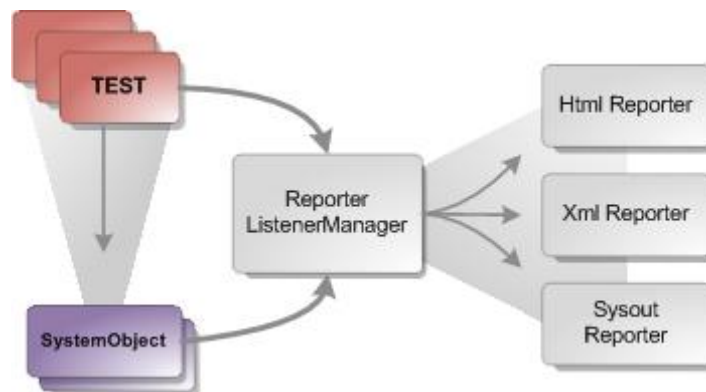


Figure 3: Service Report Structure

As is shown in the diagram the test /System Object is run and sends a report message to the **"Report Listener Manager"**, as a result the **"Reports Listener Manager"** sends the event to the HTML reporter, XML reporter and the Systout Reporter. The user can add additional reporters as required by implementing the reporter java interface and updating the JSystem properties.

Note: *The customizable reporter service enables the programmer to add reporter services to the output of the "Reporter ListenerManager".*

The reporter is a customizable service where by the programmer can customize a single reporter plug-in that enables the sending of report messages to different terminal reporters. Each reporter can then receive the reporter information and process it according to the locally configured requirements.

Unlike other reporting systems the JSystem Reporter has the ability to influence the success status of the test. That means that when a fail report message is sent the system will identify the test as a failed test.

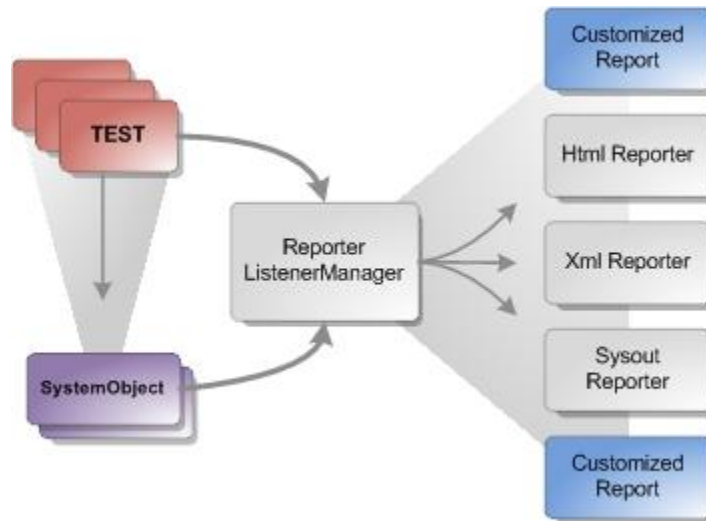


Figure 4: Customized Report Services

Reference: For more information about customizing the Reporter go to Error! Reference source not found. on page Error! Bookmark not defined..

4.5.1.1 Using the Reporter Service

When the JSystem framework is run, it automatically starts the report service. The user writes a test/system objects, within the test code the user then activates the Reporter Listeners Manager. Once the test has been written and is complete the user can now implement the test within a scenario.

The user now runs the scenario and the report service automatically runs and begins to track report events. The report events arrive to the report service and are transferred to the various Reporters as is shown in the illustration.

4.5.1.2 Test Flow Functionality

The reporter service is used by the framework as a means to connect into test flow and to enable the "**pause**" and "**graceful stop**" functions. In addition to the direct benefits of using the JReporter system that provides a rich and informative test log, it is important to use the JReporter service to generate increased behavioral test functionality by implementing the "**pause**" and "**graceful stop**" features.

4.5.1.3 Test Code Example

The code example illustrates a simple traffic test that generates traffic and then analysis it. The purpose of this piece of code is to demonstrate how the reporter usage is incorporated into the test code.

```
01  /**
02   * Simple play.
03   *
04   */
05  public void testSimplePlay() throws Exception {
06      /**
07       * Send burst of 100 packets, test that 100 packets received
08       */
09      report.report("Set transmit rate to 100 packets per second")
10      ;
11      tg.port[0].ps[0].setRatePPS(100);
12      tg.port[1].ps[0].setRate(200);
13      report.step("Set burst size to 100 packets");
14      tg.port[0].ps[0].setBurstMode(100);
15      report.step("Set port 0 to send packets");
16      tg.aPlay.setPlayers(new int[] {0});
17      report.step("Set port 1 to receive packets");
18      tg.aPlay.setReceivers(new int[] {1});
19      report.step("Set modes, play and analyze");
20      tg.aPlay.setModes(false, false, false, false);
21      tg.aPlay.play();
22      tg.aPlay.analyze(new TrafficCounter(1, EthernetPortCounters.RECIEVE_PACKETS, 100, 0.1));
23      tg.aPlay.analyze(new TrafficCounter(1, EthernetPortCounters.DATA_INTEGRITY_ERRORS, 0, 0));
24  }
```

Table 11: Reports Code Example

4.5.2 Configuring the Reporter Service

In order to configuring the reporter service the programmer must update the **"jsystem.properties"** file.

The following table lists the relevant properties.

Property	Description	Comments
logger	Enable/disable the reporter	Possible values are true/false
htmlReportDir	Output folder of the reporters	
html.old.dir	Html reports backup	
html.zip.disable	Enable/disable reports backup	Possible values are true/false
warning.color	The color of warning messages	
summary.disable	When set to true, signal the system to not aggregate events in JRunner heap memory	Should be used in long runs, to prevent memory problems
reporter.classes	Use this property to add custom reporters	You should point the reader to the section that describes how to extend the reporter
browser	The browser that will be used by the JRunner to open the html reports	

Table 12: Configuring the Reporter Service

4.6 HTML Reporter

The most commonly used reporter application for the analysis of execution results is the “**HTML Reporter**”.

The following section will demonstrate the reporter API functionality and show how the API is reflected in the HTML Reporter within a standard internet browser application.

4.6.1 Basic Reporter Methods

In order to run the JSystem Report example shown, the programmer writes the following piece of code in “**Eclipse**” and compiles the code. The user can run the code example from within either the “**Eclipse**” or “**JSystem**” environments.

4.6.1.1 Basic Reporter Code Example

```
1 /**      * Demonstrates basic reporter methods.
2      */
3     public void testReporterBasics() throws Exception{
4         //Logs simple reporter message
5         report.report("message title");
6
7         //Logs a report message with title and internal content with success status
8         report.report("message title","internal message",true);
9     }
```

Table 13: HTML Reporter Basic Code Example

4.6.1.2 Screen Shot

The HTML browser screen shot illustrates the result of the basic code example above.

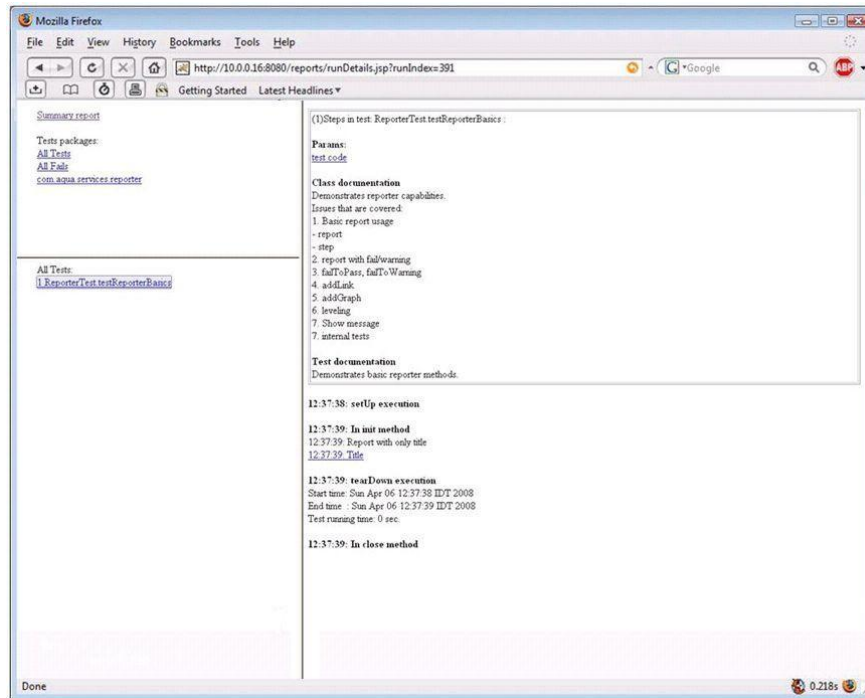


Figure 5: HTML Basic Code Test Example

The second layer of the test is accessed by pressing the highlighted “Blue” colored “Title” link as illustrated in the HTML example. The next page illustrated in the screen shot is the second layer of the test example’s output.

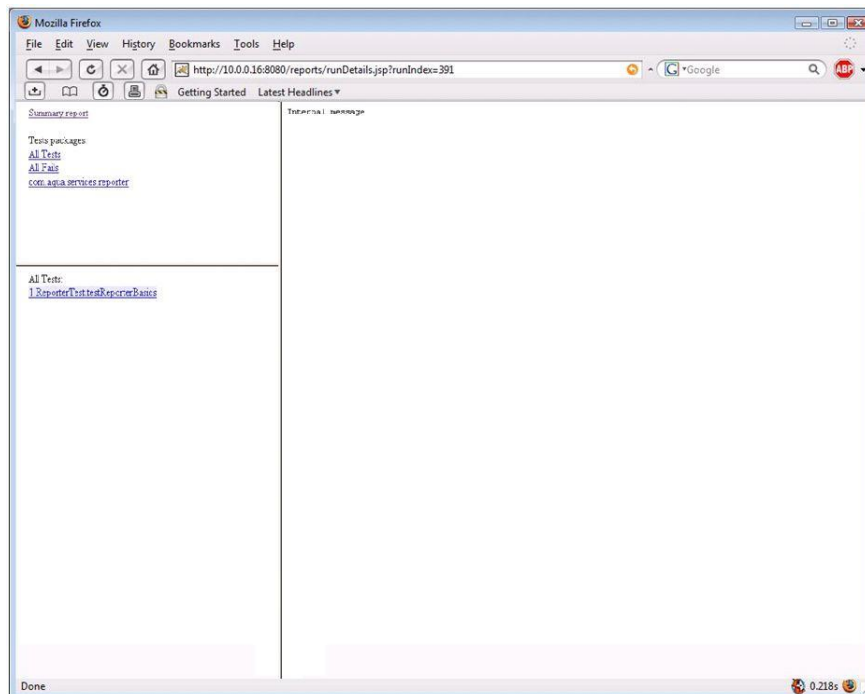


Figure 6: HTML Basic Code Level 2 Test Example

4.6.2 Changing the Test Success Status

The following code example demonstrates the implementation of either a test warning status indicator or a test fail status indicator in the HTML browser application after compilation.

4.6.2.1 Test Status Code Example

```
1      * Demonstrates reporter message which causes test to
2      * fail or to be in warning status
3      */
4      public void testReporterWithErrors() throws Exception{
5          report.report("report with error", "internal message", Reporter.FAIL /** Reporter.WARNING */);
6      }
```

Table 14: Test Success Status Code Example

4.6.2.2 JSystem Screen Shot

The screen shot illustrates the result of the “Warning” and “Fail” code examples.

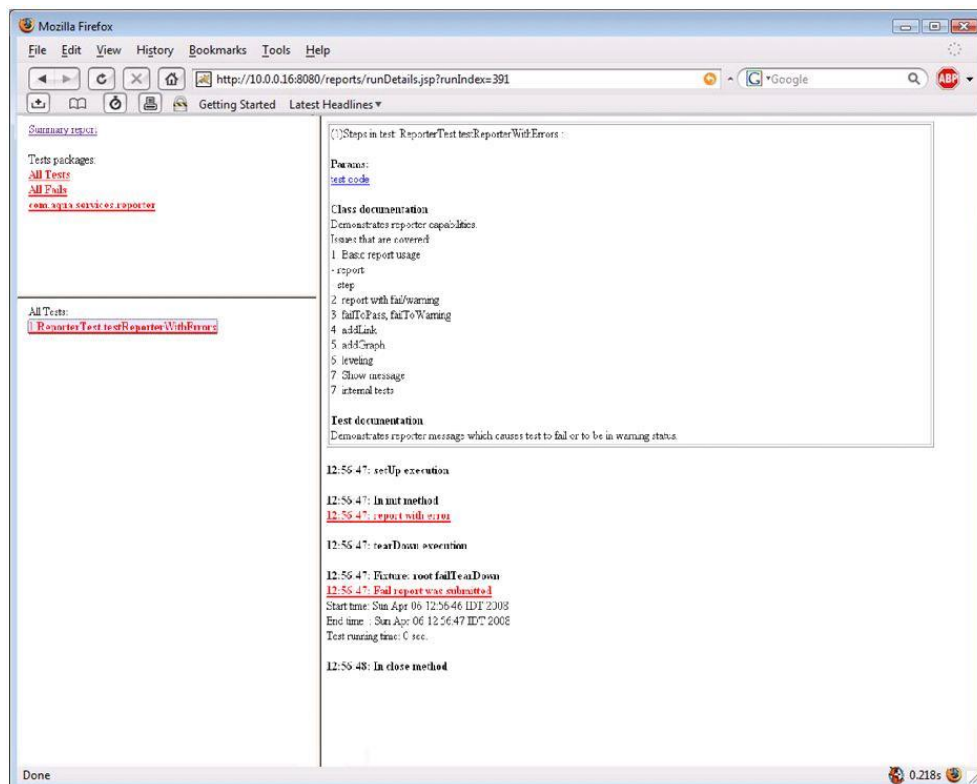


Figure 7: Success Status HTML Reporter Example

4.6.3 Logging Exceptions

The following code example demonstrates the implementation of a test fail status indicator with an exception status indicator in the HTML browser application after compilation.

4.6.3.1 Logging Exceptions Code Example

```
1  /**
2   * Demonstrates reporter message which causes test to fail with an exception.
3   */
4   public void testReporterWithException() throws Exception{
5       report.report("report with exception",new Exception("An error"));
6   }
```

Table 15: Failure with an Exception Code Example

4.6.3.2 JSystem Screen Shot

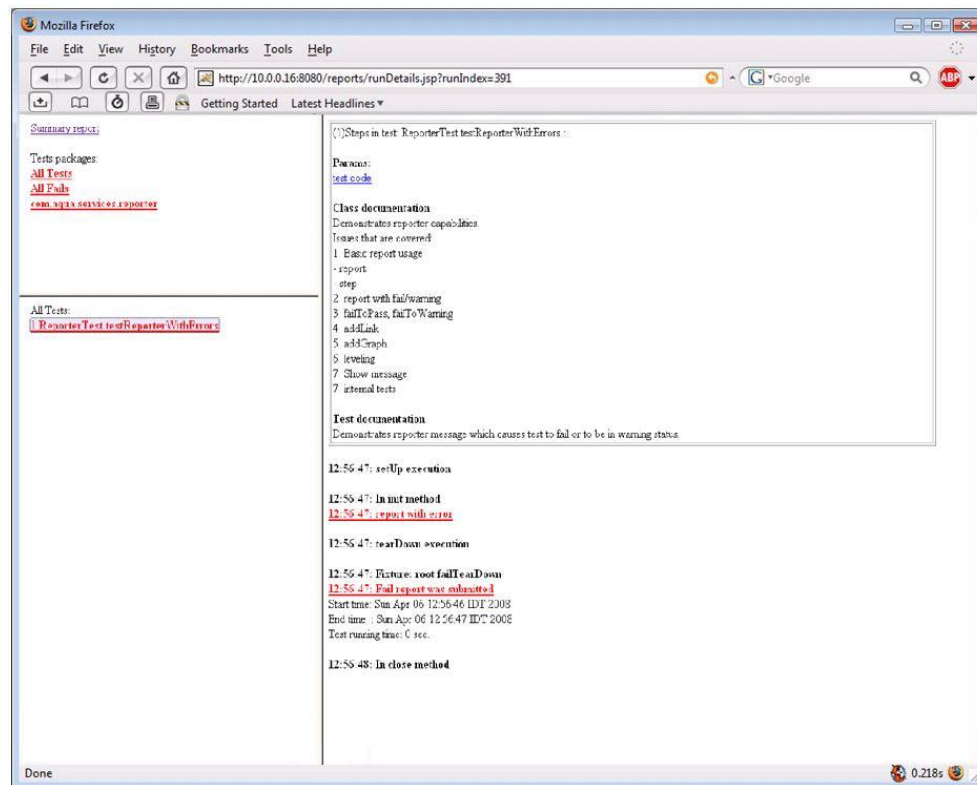


Figure 8: Failure with an Exception HTML Example

By pressing the “**report with exception**” link the user accesses the second layer of the report that illustrates the exception.

4.6.3.3 HTML Report Screen Shot

The snapshot shows the HTML report when drilling down into the exception.

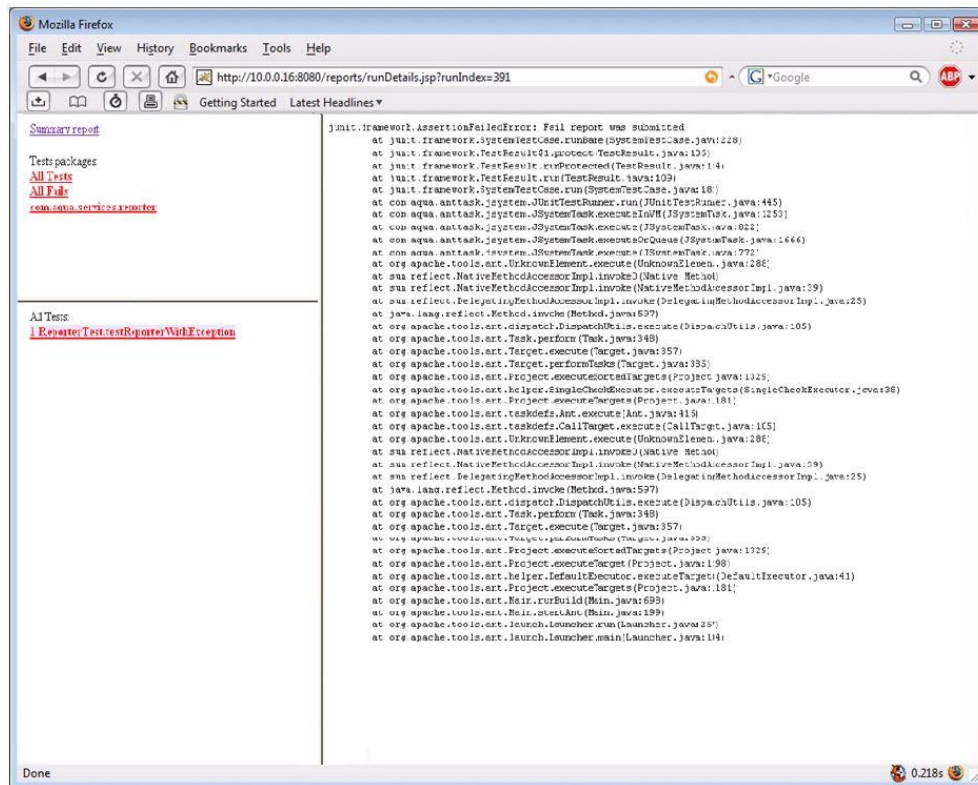


Figure 9: HTML Exceptions Example

4.6.4 Steps

The code example demonstrates the use of steps; steps are reflected in two places, they are saved in the JReporter application database, and displayed in the test summary report in the JReporter application.

4.6.4.1 Steps Code Example

```
01  /**
02   * Demonstrates step usage.
03   * Steps are reflected in two places:
04   * 1. Steps are saved in the Jsystem reports application da
05   *    tabase, and reports which show a summary
06   *    of all the steps in the test can be generated from th
07   *    is information.
08   * 2. Steps are showed in bold format in the HTML reporter
09   *    and Jsystem reporter tab.
10   */
11  public void testReporterStep() throws Exception{
12      report.step("step description");
13  }
```

Table 16: Steps Code Example

4.6.4.2 Steps Screen Shot

The screenshot shows the HTML report that is reproduced by the **"step"** method.

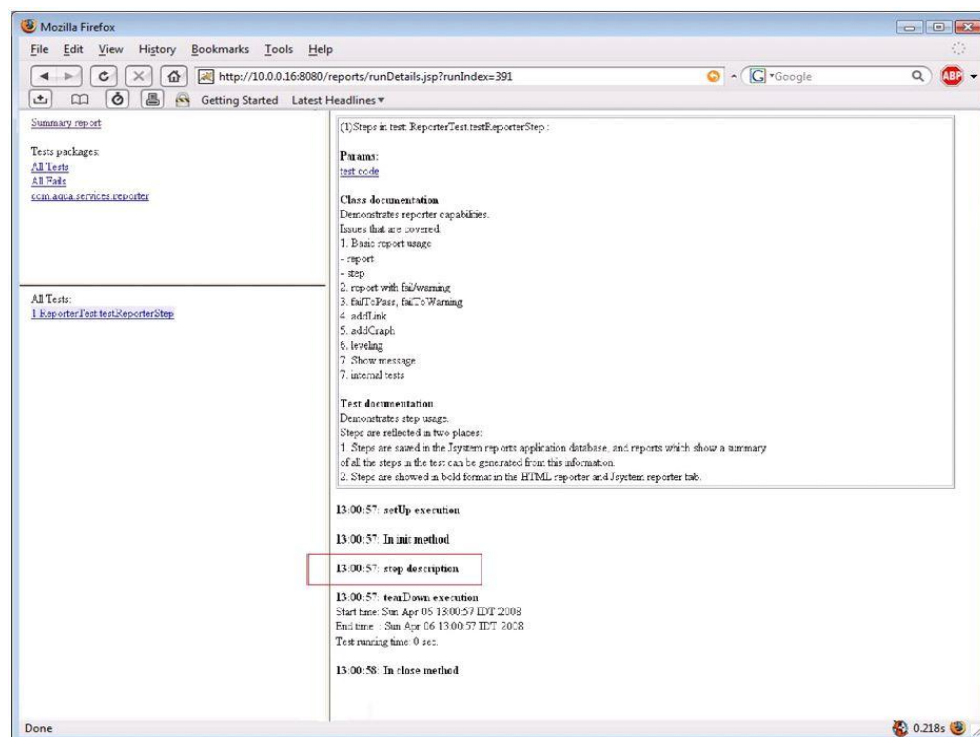


Figure 10: Steps HTML Browser Example

4.6.5 Links

It is important to note that in order to produce output from the JReporter in the form of a self contained file that the programmer wishes to link, it should be copied as follows.

Note: *In order to perform this function there is a utility class that copies the file to the reporter folder adding a link to the file.*

4.6.5.1 Links Code Example

The code example demonstrates linking a file in the reporter.

```
1  /**
2   */
3   public void testReporterAddLinkToFile() throws Exception{
4       File f =station.getFile("myFile.txt");
5       ReporterHelper.copyFileToReporterAndAddLink(report, f, "Li
6   nk to file");
7   }
```

Table 17: Links Code Example

4.6.5.2 Add Link Screen Shot

The screen shot shows the HTML report of the **"testReporterAddLinkToFile"** test.

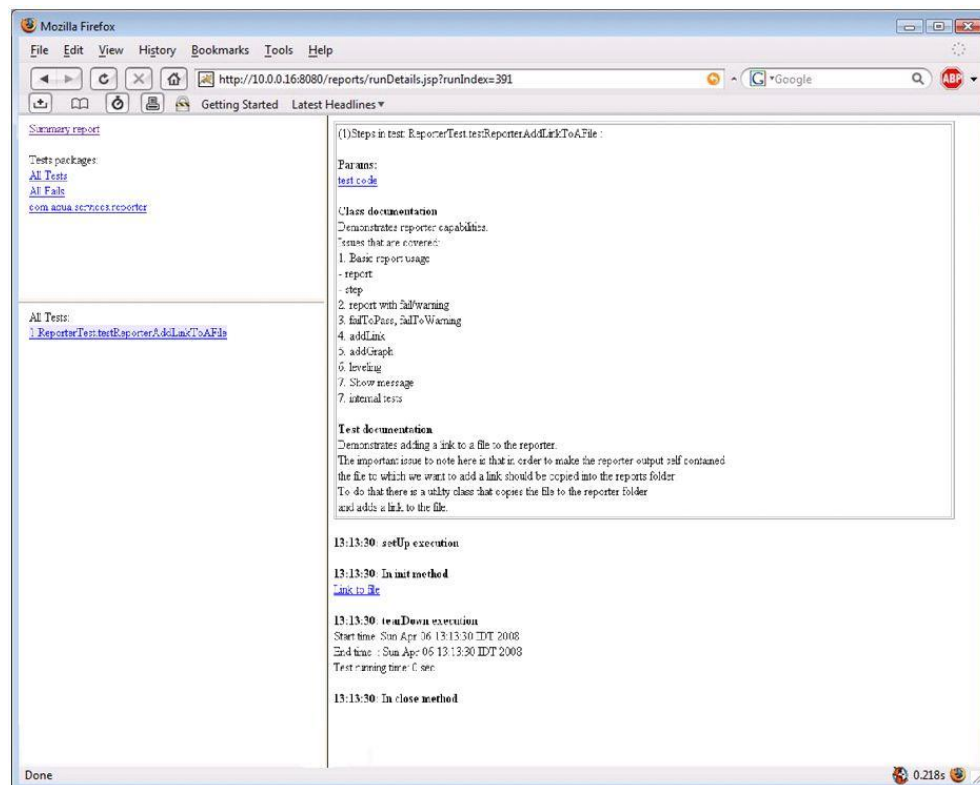


Figure 11: Links HTML Example

4.6.5.3 Link to File Screen Shot

The screen shot shows the HTML page that opens after pressing the “**Link to file**” link.

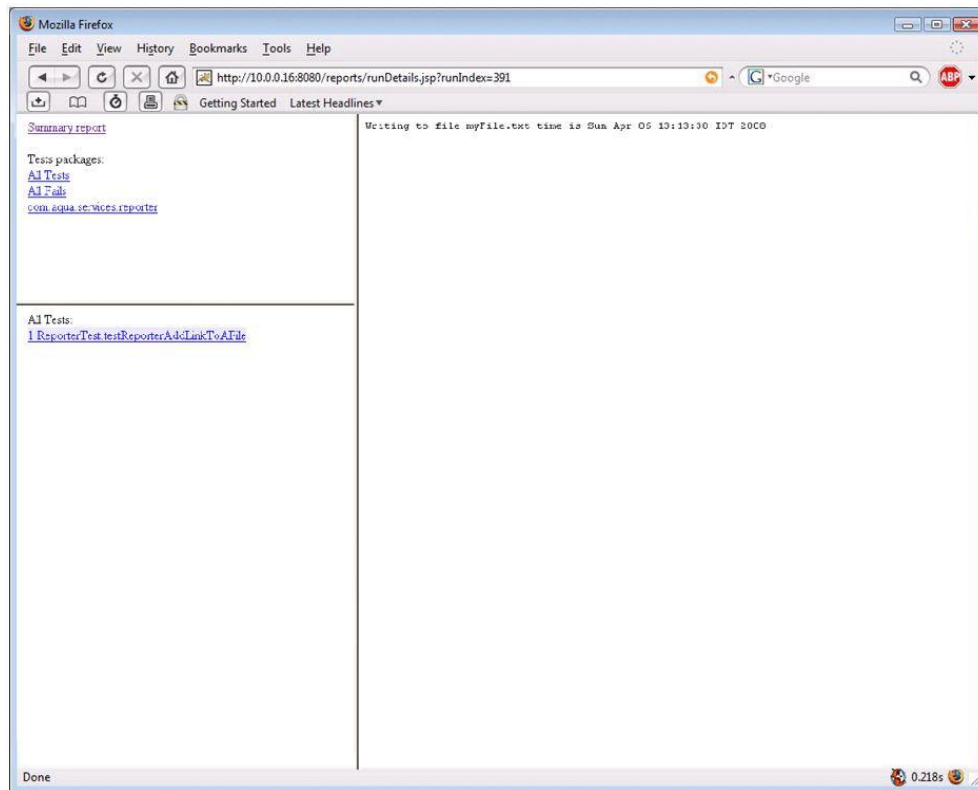


Figure 12: Links HTML 2nd Level Example

4.6.6 Graphs

This piece of code demonstrates the integration of an HTML visual graph element to the report structure.

4.6.6.1.1 Code Example

```
01  /**
02   * Demonstrates report with graph.
03   */
04   public void testReportWithAGraph() throws Exception{
05       Graph graph = GraphMonitorManager.getInstance().allocate
Graph(
06           "Dummy graph of number of recieved packets in ping o
peration over time", "received packets");
07       for (int i = 0; i < 10;i++) {
08           int receive = new Random().nextInt();
09           graph.add("receive", i, receive);
10       }
11       graph.show(report);
12   }
```

Table 18: Test Report Graph Code Example

4.6.6.1.2 Graph Screen Shot

The compiled code executes the following HTML graph within the JReporter.

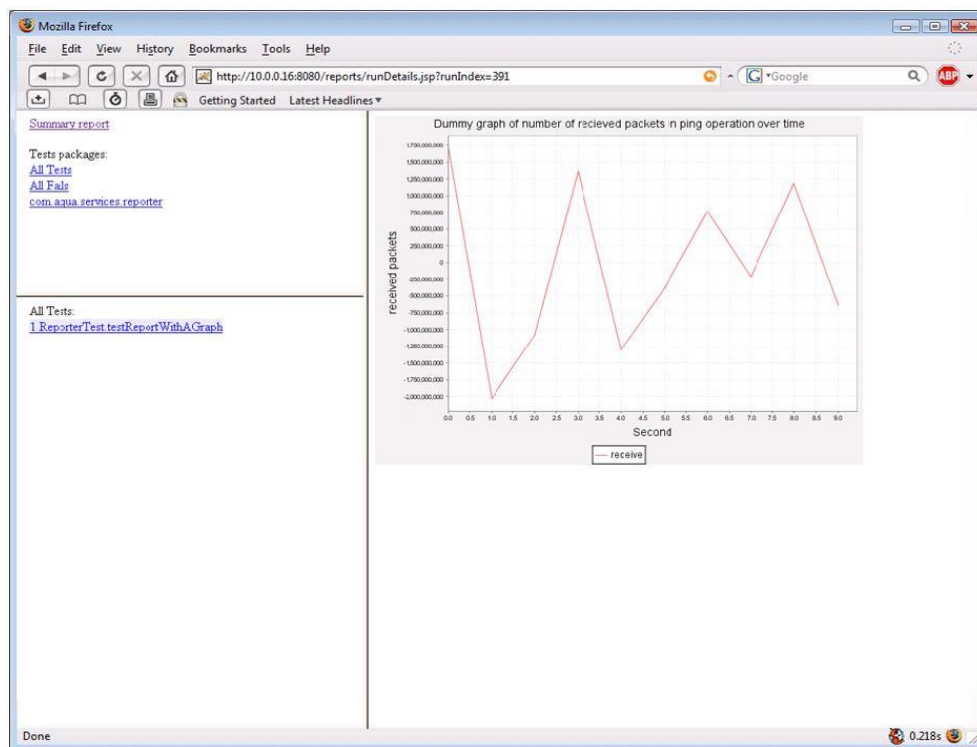


Figure 13: Graph HTML Report

4.6.7 Levels

The report API enables the programmer to create a hierarchical report where by each level contains a link to the next level. By implementing the leveling function the user can build clean focused reports.

4.6.7.1 Levels Code Example

Each level of the report contains a link to the next level, using the leveling code example, the user can build cleaner and more focused reports.

```
01 /**
02  */
03  public void testReportWithLeveling() throws Exception{
04      report.startLevel("first level", Reporter.MainFrame);
05      report.report("message in level 1");
06      report.startLevel("second level", Reporter.CurrentPlace);
07      report.report("message in level 2");
08      report.startLevel("third level", Reporter.CurrentPlace);
09      report.report("message in level 3");
10      report.stopLevel();
11      report.report("another message in level 2");
12      report.stopLevel();
13      report.report("another message in level 1");
14      report.stopLevel();
15      report.report("message in main report page");
16  }
```

Table 19: Report API Code Example

4.6.7.2 Test Report Levels Screen Shot

The screen shot of the HTML report shows the **"testReportWithLeveling"** test.

The main page of the test report has a link in it **"first level"** by pressing it the user will be directed to the test's first level page.

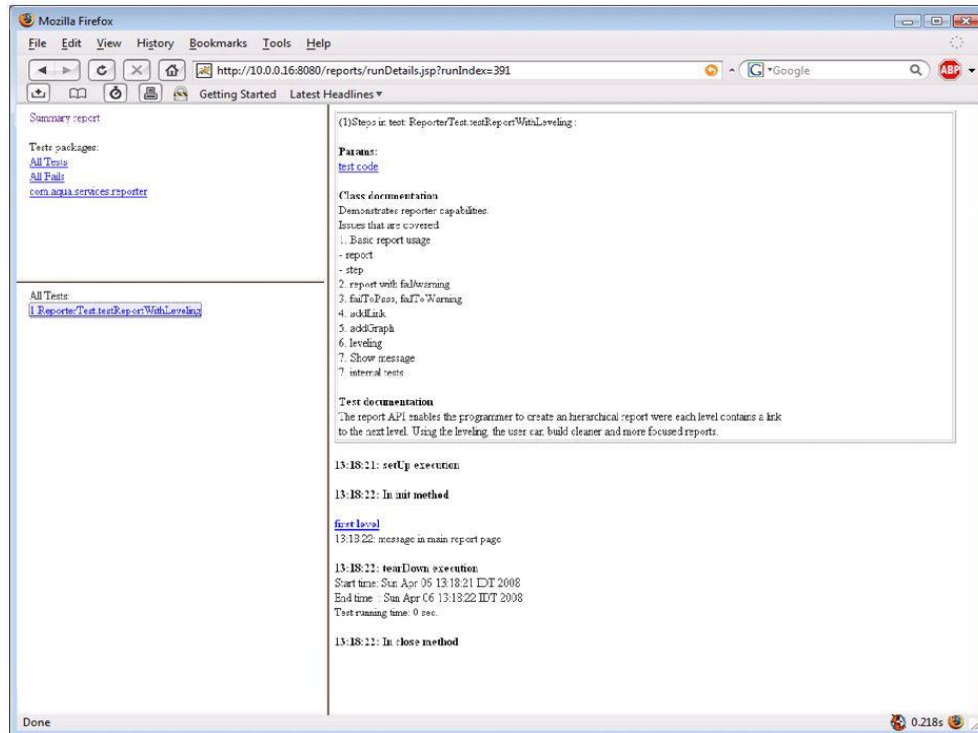


Figure 14 : Report API HTML Example

The snapshot shows the **"first level"** web page. Notice that the **"first level"** page has a link to **"second level"**. The user has the flexibility to create as many levels as is required.

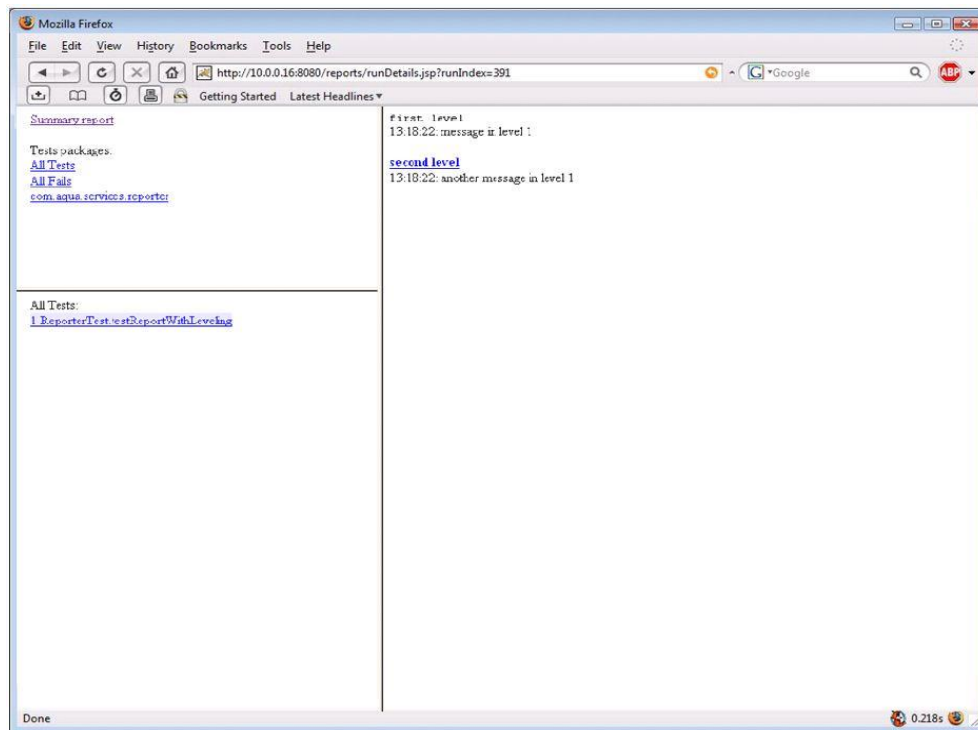


Figure 15: Report API HTML 2nd Level Example

4.6.8 Interactive Messages

The code method illustrates the usage of pop-up interactive messages from within the reporter API.

4.6.8.1 Interactive Messages Code Example

This method shows how to add a pop up interactive messages to the user interface using the reporter API.

```
1  /**
2   public void testInteractiveMessage() throws Exception{
3       int res = report.showConfirmDialog("Confirm Dialog","Conti
4       nue test?",JOptionPane.YES_NO_OPTION,JOptionPane.INFORMATION_MESS
5       AGE);
6       report.step("user reply = " + res);
7   }
```

Table 20: Reporter API Popup Code Example

4.6.9 JSystem Snapshot

The snapshot shows the JRunner when the interactive message is processed.

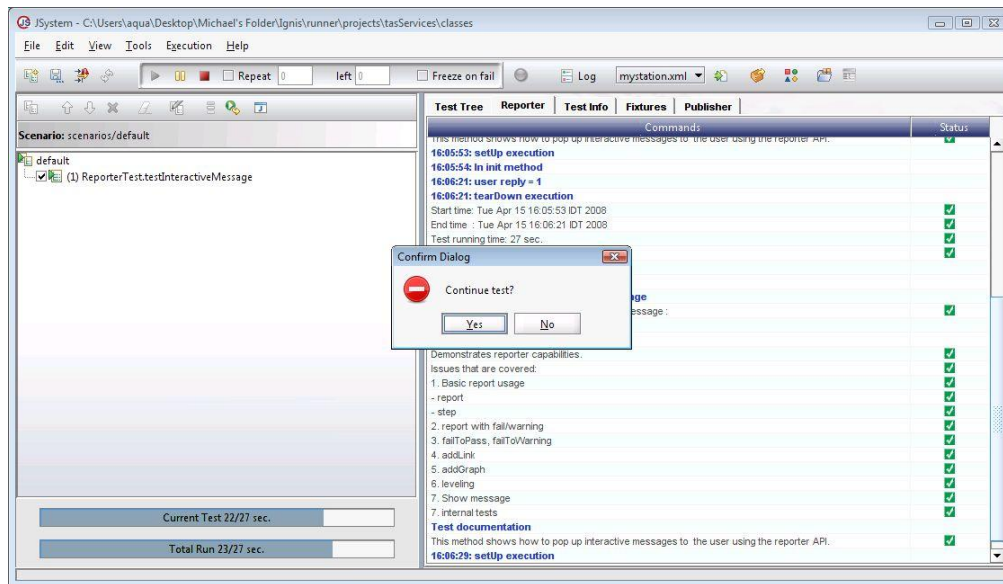


Figure 16: Reporter Popup JRunner Example

4.7 Analyzers

Analyzers provide an easy way to verify “**SystemObjects**” (driver) operations. The system object operations are used to manage and or receive information from the SUT. Every operation takes the result object and defines it as the object to be analyzed using the “**setTestAgainstObject**” method. Thus, the analyzers can be used to analyze the object and compare it with an expected result.

The analysis service comprises an architecture that supports two kinds of users, an advanced programmer that writes the system object and analyzers, and users that use the analyzers to in order to examine operation results.

Predefined analyzers provide an easy method to verify system object operations. Every system object operation takes the command output and sets it to be the “**testAgainst**” object.

4.7.1 Using Analyzers

The components that comprise the analysis process are as follows:

1. The system object that performs the operation on the SUT, and defines the received results of the operation on the system object that will be performed by the analysis.
2. The test that activates the analyzer on received operation results.
3. The analyzer java class that perform the actual analysis.

4.7.1.1 Simple Traffic

The code example illustrates a simple traffic test that generates traffic and then analysis it. The purpose of this piece of code is to demonstrate how analyzers are used.

4.7.1.2 Simple Traffic Code Example

In the code example “**testIsFileExists**”, the programmer activates the “**dir**” operation of the system object. The result of the operation defines the object on which the analysis is performed.

```
01  /**
02   * Checks whether file <code>fileToFind</code> exists in
03   * <code>folder</code>.
04   * Demonstrates basic analysis usage
05   * @params.include fileToFind,folder
06   */
07   public void testIsFileExists() throws Exception {
08       station.dir(getFolder());
09       station.analyze(new SimpleTextFinder(getFileToFind()));
10 }
```

Table 21: Simple Traffic Code Example

4.7.2 Analyzer Methodology

In order to perform an analysis the programmer activates the analyzer using the analyze method.

If the analysis performed is successful the reporter service will produce a positive result. If unsuccessful an error indication will be reported and the system object will throw an assertion exception.

4.7.2.1 Directory Operation Code Example

In the code example the stations system object uses the Cli system object to send the “**dir**” command, it then takes the command results and defines them to be the data on which the analysis is performed, this is achieved by activating the “**setTestAgainstObject**” method.

```
01 public void dir(String folderName) throws Exception {  
02     CliCommand cmd = new CliCommand("dir " + folderName);  
03     cmd.addErrors("unknown command");  
04     cliConnection.handleCliCommand("dir " + folderName, cmd);  
05     setTestAgainstObject(cmd.getResult());  
06 }
```

Table 22: Directory Operation Code Example

4.7.3 Writing an Analyzer

In the two previous sections the user learned about analyzer functionality. Now that the analyzer has been defined the user must now learn how to implement an analyzer. In order to implement an analyzer the user must extend the "**AnalyzerParameterImpl**" class. This is achieved by performing the following implementation of an analyzer method as shown in the following example:

4.7.3.1 Analyzer Implementation Guidelines

The following Guidelines refer to analysis "**() method**" implementation:

1. In order to signal the JSystem framework if the analysis has failed or succeeded, the programmer must set the status class member field to a true or false value. False = **failure** and true = **success**.
2. Implementing the title and message fields. The "**title**" class member should be set with the summary of the analysis results. The "**message**" class member is usually used in the case of an analysis failure and is set with the "**toString()**" of operation results.

```
01 /**
02  * Simple analyzer example
03  * @author goland
04  */
05 public class SimpleTextFinder extends AnalyzerParameterImpl {
06     private String txtToFind;
07     int result;
08     public SimpleTextFinder(String txtToFind){
09         this.txtToFind = txtToFind;
10     }
11     public void analyze() {
12         String txt = testAgainst.toString();
13         if (StringUtils.isEmpty(txt)){
14             title = "No text was given to analyzer";
15             status= false;
16             return;
17         }
18         if (txt.indexOf(txtToFind) > 0){
19             title = "Text " + txtToFind + " was found";
20             status = true;
21             return;
22         }
23         title = "Text " + txtToFind + " was not found found";
24         message = txtToFind;
25         status = false;
26         result = 5;
27     }
28     public int getResult(){
29         return result;
30     }
31 }
```

Table 23: Simple Analyzer Code Example

4.7.4 Advanced Analyzers Features

Default usage of the analyze method throws an “**AnalyzerException**” java class and issues an error report when an error is received.

There are two options that the programmer can implement in order to prevent JSystem from throwing an exception. They are as follows:

1. Before analysis, activate the method “**station.setThrowException(false);**”
2. In order to see the method documentation use the following analysis method “**station.analyze(analyzer,false,false);**”

```
01 /**
02
03  * Analyze the result. Report analysis results to the reporter
04  *
05  * @param parameter
06  *           The analyzer parameter object.
07  * @param silent
08  *           if silent no reports will be submitted
09  * @param throwException
10  *           if false an AnalyzerException will not be thrown in case of analysis failure
11  *
12  * @exception AnalyzerException
13  */
14 @SuppressWarnings("unchecked")
15 public void analyze(AnalyzerParameter parameter, boolean silent, boolean throwException) throws AnalyzerException {
```

Table 24: Advanced Analyzer Report Result Code

4.7.4.1 Prevent Publishing Error Reports

In order to prevent the analyzer from throwing exceptions and not publishing error reports to the reporter the programmer must implement the following method, “**isAnalyzeSuccess(AnalyzerParameter parameter)**”. This method activates the Analyzer and returns the analysis results.

4.8 Monitors

Monitors are processes that are also referred to as threads; these threads are defined per test. The monitor thread process runs during the test execution via a separate thread. The monitor can fail the test if it finds a problem with the monitored object.

The monitor thread is a parallel thread to the test itself that enables the user to monitor the device under test (DUT). The service that the framework provides is the ability to control the life cycle of the monitor.

4.8.1 Writing a Monitor

In order to implement a monitor the programmer has to extend the monitor class and implement monitor logic in the **"run()"** method.

4.8.1.1 Monitor Class Code Example

The example illustrates a monitor class; the run method of the monitor class sends a request to the framework for the station system object. The system object then activates a ping operation from the machine represented by the station system object to the address provided by the ping monitor, after performing ping the monitor analyzes ping results.

```
01 package com.aqua.services.demo;
02
03 import jsystem.framework.monitor.Monitor;
04
05 public class PingMonitor extends Monitor {
06     String pingTo;
07     public PingMonitor(String pingTo) {
08         super("ping monitor");
09         this.pingTo = pingTo;
10     }
11
12     public void run() {
13         WindowsStation station;
14         while (true) {
15             try {
16                 station = (WindowsStation)
17                     system.getSystemObject("station");
18                 station.ping(pingTo);
19                 station.analyze(new FindText("0% loss"))
20                 Thread.sleep(1000);
21             } catch (Exception e) {
22                 return;
23             }
24         }
25     }
26 }
```

Table 25: Monitor Class Code Example

4.9 Fixtures

One of the most important guidelines in automation development is separating configuration code from testing code. Separating the code allows for reusable configuration code, producing tests that are simple to read and write. These simple tests should be written as independent stand alone tests that are not configured with interdependent characteristics within a single scenario.

Separating test code from configuration code is done by writing fixtures. The fixture is a special type of test class that brings the SUT to a specific configuration state.

Note: A naive separation between the configuration and testing environments that keeps on testing independently with reoccurring configurations.

Warning: The result is a waste of time and in many cases wears out the device being tested.

4.9.1 Fixtures Module

In addition to supporting separation of configuration code from testing code, JSystem's fixtures module eliminates inefficient and time consuming testing practices and needless device wear and tear by performing the following functions:

- A fixture is a special type of test class that brings the SUT to the desired state
- Each fixture performs an SUT update and or configuration operation, and should know how to rollback and or cleans up after an operation has been performed.
- Fixtures behave in a symbiotic relationship to one and other and can therefore be ordered in to a hierarchical structure.
- Each test associate itself with a fixture. While performing a runtime, the framework automatically navigates to the fixture which is required by the test.

4.9.2 Writing a Fixture

JSystem fixtures are used to extend the “**jssystem.framework.fixture.Fixture**” class and are required in order to implement a setup method an optional “**tearDown**” method and a “**failTearDown**” method.

4.9.2.1 Fixture Class Code Example

```
01 package com.aqua.services.fixtures;
02 import jssystem.framework.fixture.Fixture;
03 public class ExampleFixture extends Fixture {
04     public ExampleFixture() {
05     }
06     public void setUp() throws Exception {
07         report.step(" in example fixture setup");
08     }
09     public void tearDown() throws Exception {
10         report.step(" in example fixture tearDown");
11     }
12     public void failTearDown() throws Exception {
13     }
14 }
```

Table 26: Writing a Fixture Code Example

The setup method is activated when navigating to the fixture; the “**tearDown**” method is activated when navigating from the fixture to another fixture. The “**failTearDown**” method is activated when navigating to another fixture after test failure.

Reference: For more information regarding fixture navigation see *Fixture Navigation*, on page 43.

4.9.3 Creating Fixtures Hierarchy

In order to create a fixture hierarchy the programmer must set the fixture's parent by calling the "**setParentFixture**" method in the fixture constructor.

```
01 package com.aqua.services.fixtures;
02 import jsystem.framework.fixture.Fixture;
03 public class ExampleFixture extends Fixture {
04     public ExampleFixture() {
05         setParentFixture(ParentExampleFixture.class);
06     }
07     public void setUp() throws Exception {
08         report.step(" in example fixture setup");
09     }
10     public void tearDown() throws Exception {
11         report.step(" in example fixture tearDown");
12     }
13     public void failTearDown() throws Exception {
14     }
15 }
```

Table 27: Creating Fixture Hierarchy Code Example

Note: When the programmer creates a loop in the fixtures hierarchy ($F1 \rightarrow F2 \rightarrow F3 \rightarrow F1$), the system identifies the loop and as a result the test execution fails.

4.9.4 Associating a Test with a Fixture

In order to signal the system to navigate to a certain fixture before running the test in test class constructor, use the “**setFixture**” method.

```
01 package com.aqua.services.fixtures;
02 import junit.framework.TestCase;
03 public class FixturesDemonstrationTest extends TestCase {
04     public FixturesDemonstrationTest() {
05         setFixture(ExampleFixture.class);
06     }
07     public void testFixturesExample() {
08         report.step("----- in test");
09     }
10     public void testAnotherFixturesExample() {
11         report.step("----- in test");
12     }
13 }
```

Table 28: Associating a Test with a Fixture Code Example

When automatically associating a test class with a fixture all of the test cases in the class are associated with the fixture. This behavior can be overridden by associating a fixture to a scenario.

4.9.5 Fixture Navigation

This section describes how the system navigates between fixtures. The following illustration shows the process flow and structure of the fixture tree.

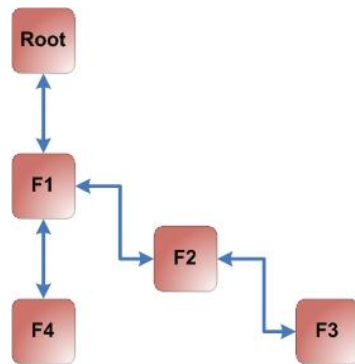


Figure 17: Fixture Hierarchy Structure

4.9.5.1 Test and Fixture Association

The table shows the relationship between the tests and their corresponding fixtures. The explanation that follows explains the relationship between the corresponding tests and scenario fixtures.

Test	Fixture	Description
	(Fixture 2)	The JRunner executes the scenario T1 (Root) and searches for its associated fixture. After determining that T1 is associated to the fixture F2, it navigates to F2 via F1. The navigation includes running the setup method of F1 and setup method of F2.
(Test 2)	(Fixture 3)	After navigating to F2 the fixture navigator activates the test T1. After finishing T1 it checks which fixture T2 it is associated to, the fixture is F3 so it navigates from F2 (the current fixture) to F3. This activation includes activating the setup method of F3.
(Test 3)	(Fixture 4)	After running the T2 test the fixture navigator moves to T3. The fixture which is associated with T3 is F4 so now the fixture navigator has to navigate from F3 to F4. The navigation includes moving from F3 to F2 and from F2 to F1 and from F1 to F4. So to move from F3 to F2 is activates the tearDown method of F3 then the tearDown method of F2, it does not activate the tearDown of F1 but goes to setup method of F4.

Table 29: Scenario Fixture Association

4.9.6 Tear Down Fixture

In the event of a **“test failure”** where the programmer needs to perform a **“cleanup”** process that differentiates itself from a standard **“failtest”** process the following teardown fixture is applied, as appears in the code example.

4.9.6.1 Fail Tear Down Code Example

If the user defines a tear-down fixture for a test and the test fails, the fixtures navigator navigates to the tear down fixture.

When navigating to a tear down fixture, the fixture navigator looks for a **“failTearDown”** method in the fixture.

```
01 package com.aqua.services.fixtures;
02 import jsystem.framework.fixture.Fixture;
03 public class ExampleFixture extends Fixture {
04     public ExampleFixture() {
05         setParentFixture(ParentExampleFixture.class);
06     }
07     public void setUp() throws Exception {
08         report.step(" in example fixture setup");
09     }
10     public void tearDown() throws Exception {
11         report.step(" in example fixture tearDown");
12     }
13     public void failTearDown() throws Exception {
14     }
15 }
```

Table 30: Tear Down Fixture Code Example

Note: In order to support a “failTearDown” method it needs to be added to the fixture.

4.9.6.2 Test Constructor Code Example

In the test the constructor defines the fail down fixture.

```
01 package com.aqua.services.fixtures;
02 import jsystem.framework.fixture.RootFixture;
03 import junit.framework.SystemTestCase;
04 public class FixturesDemonstrationTest extends SystemTestCase {
05     public FixturesDemonstrationTest() {
06         setFixture(ExampleFixture.class);
07         setTearDownFixture(RootFixture.class);
08     }
09     public void testFixturesExample() {
10         report.step("----- in test");
11     }
12 }
```

Table 31: Test Constructor Code Example

4.9.6.3 Navigating with a TearDown Fixture

Enhancing the fixtures navigation example in order to see how it works with a fail down fixture.

Test	Fixture	Tear Down Fixture
T1	F2	
T2	F3	F1
T3	F4	

Table 32: Fixture Navigation Examples

In the example the teardown fixture of T2 has been defined as F1. During the scenario execution in the event that T2 fails the fixture navigator will then navigate from F3 to F1 in the following way:

1. The fixture will activate the "**failTeardown**" of F3, and then it will activate the "**failTeardown**" of F2.
2. Before execution the T3 fixture will navigate to the F4 fixture by activating the setup method of the F4 fixture.

4.9.7 Working with Fixtures in the JRunner

The Fixtures tab in the JRunner shows the fixtures that are defined in the system and their resulting hierarchical structure:

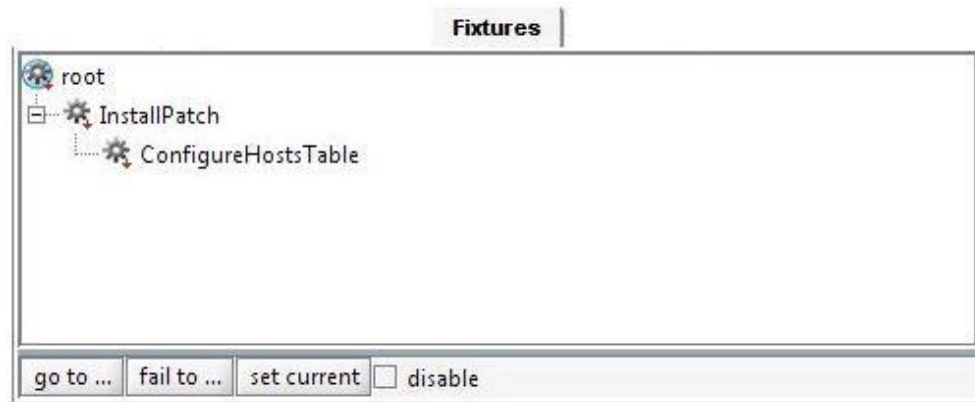



Figure 18: JRunner Fixture View

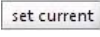
During the execution of a scenario, the JRunner graphically navigates down the fixtures tree inside the fixtures tab. The current fixture is highlighted in blue.

4.9.7.1 Manual Navigation to a Fixture

In order to manually navigate to a fixture press the  **"Go to"** button.

To navigate through to the **"failTearDown"** methods, press the  **"Fail to"** method as appears in the JRunner image.

4.9.7.2 Setting the Current Fixture without Navigation

In order to manually set the current fixture without having to navigate to it, select the fixture and press the  **"set current"** button.

4.9.7.3 Associating a Fixture with a Scenario

In addition to associating a fixture to a test programmatically, the JRunner user can associate a fixture with a scenario:

The illustrations below demonstrate the steps required to perform the fixture association.

1. Select the fixture from the “Test Tree” tab.

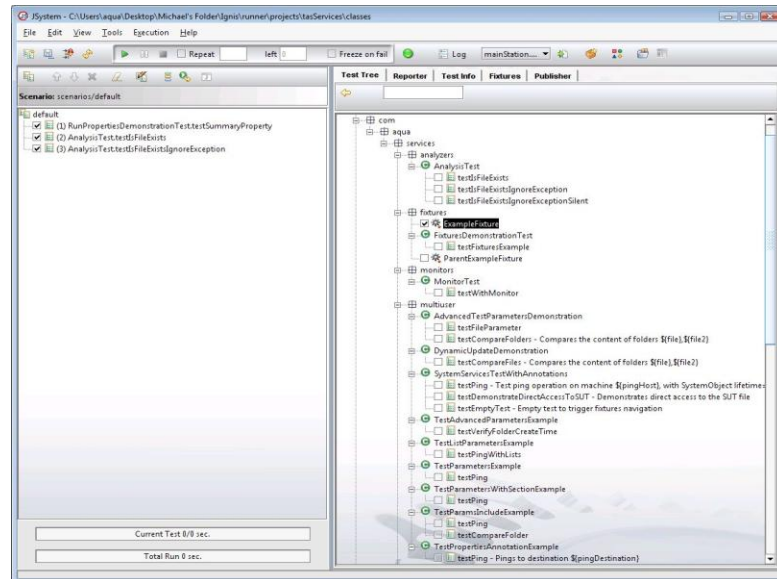


Figure 19: Associating a Fixture with a Scenario

2. Press the ➡ “add tests” button in order to add the fixture to the scenario.

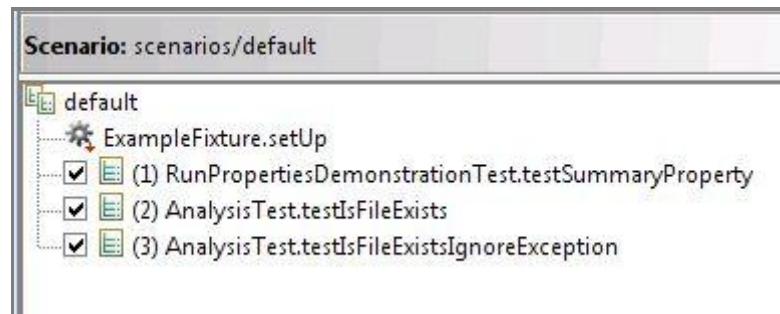


Figure 20: Associating a Fixture with a Scenario 2

Note: When associating a fixture with a scenario the fixture is associated with all the tests under it. Associating fixture to a scenario using the JRunner overrides programmatic association of fixture to a test case.

4.10 Multi-User Support

One of the most important and unique features of the JSystem Automation Platform environment is the fact that it is an automation framework providing service solutions for a varied range of users.

The first type of user is the java developer who builds the system objects and writes java tests. The second type of user is the experienced QA engineer, familiar with the system under test (SUT) but with limited or no knowledge in java programming. This user can take the building blocks that were written by the programmer and build complex scenarios.

The test author has control over the user's JRunner experience by authoring the scenarios. This is achieved by exposing tests parameters in the JRunner and controlling how they are grouped in tabs, how they are controlled and their order in the list. The test author has the ability to enable and or disable parameters by editing them.

The following section discusses how the tests author implements enhancements and additional capabilities to the scenarios editor.

4.10.1 Defining Test Parameters

The test parameters are “**test class members**”; these test class members contain setters and getters.

4.10.1.1 Code Example

```
01 package com.aqua.services.multiuser;
02 import junit.framework.TestCase;
03 /**
04  * Test parameters simple example.
05  * @author goland
06  */
07 public class TestParametersExample extends TestCase {
08
09     private String pingDestination;
10     private int packetSize;
11     public void testPing() throws Exception{
12         report.report("Test ping");
13     }
14     public String getPingDestination() {
15         return pingDestination;
16     }
17     /**
18      * Destination of ping operation.
19      */
20     public void setPingDestination(String pingDestination) {
21         this.pingDestination = pingDestination;
22     }
23     public int getPacketSize() {
24         return packetSize;
25     }
26     /**
27      * Size of packet which will be sent in ping message.
28      */
29     public void setPacketSize(int packetSize) {
30         this.packetSize = packetSize;
31     }
32 }
```

Table 33: Defining Test Parameters Code Example

4.10.1.2 JSystem JRunner GUI Representation

The “**Defining Test Parameters**” code example, as it appears in the JSystem JRunner.

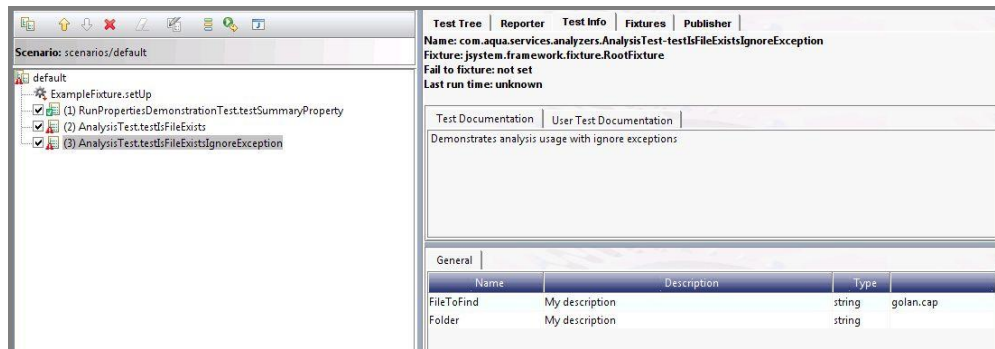


Figure 21: Defining Test Parameters HTML Example

The screen shot shows the default scenario setting in the left-hand scenarios panel of the JRunner, the “**default**” scenario contains one test, a “**pingTest**”.

In order to access the test details “**double click**” on the test, the “**Test Info**” tab becomes active and appears in the forefront within the JRunner.

The parameters panel appears within the “**Test Info**” tab and in it the two parameters that we have defined in our test are visible.

***Note:** The setter javadoc for the parameter definitions appears in the description column of the “general” tab under the “Test Tree” tab in the JSystem JRunner.*

4.10.2 Test Parameters Types

JSystem supports the following test parameter types:

- Primitive
- Close Lists
- java.util.File
- java.util.Date

4.10.2.1 Primitives

JSystem Supported Primitives - float, double, long,ini, Boolean, String.

4.10.2.2 Predefined List of Values

JSystem supports a “**Predefined List of Values**” for parameters in following two ways:

- By defining parameter type to be an enum.
- By defining parameter type to be String and adding the method `public String[] getParamNameOptions(){..}` where ParamName is the name of the parameter.

4.10.2.3 List Parameters Code Example

```
01 package com.aqua.services.multiuser;
02 import junit.framework.TestCase;
03 /**
04  * Test list parameters example.
05  * @author goland
06  */
07 public class TestListParametersExample extends TestCase {08
enum PacketSize {
09     SMALL,
10     MEDIUM,
11     BIG;
12 }
13 private String pingDestination;
14 private PacketSize packetSize;
15 public void testPingWithLists() throws Exception{
16     report.report("Test ping");
17 }
18 public String getPingDestination() {
19     return pingDestination;
20 }
21
22 public String[] getPingDestinationOptions(){
23     return new String[]{"127.0.0.1","localhost"};    }
24 public void setPingDestination(String pingDestination) {
25     this.pingDestination = pingDestination;
26 }
27 public PacketSize getPacketSize() {
28     return packetSize;
29 }
30 public void setPacketSize(PacketSize packetSize) {
31     this.packetSize = packetSize;
32 }
33 }
```

Table 34: Test Parameter Types Code Example

4.10.2.4 JSystem JRunner GUI Representation

The **"PingDestination"** parameters value is visualized as a drop down list. The list values that are returned by the method are **"getPingDestinationOptions()"**.

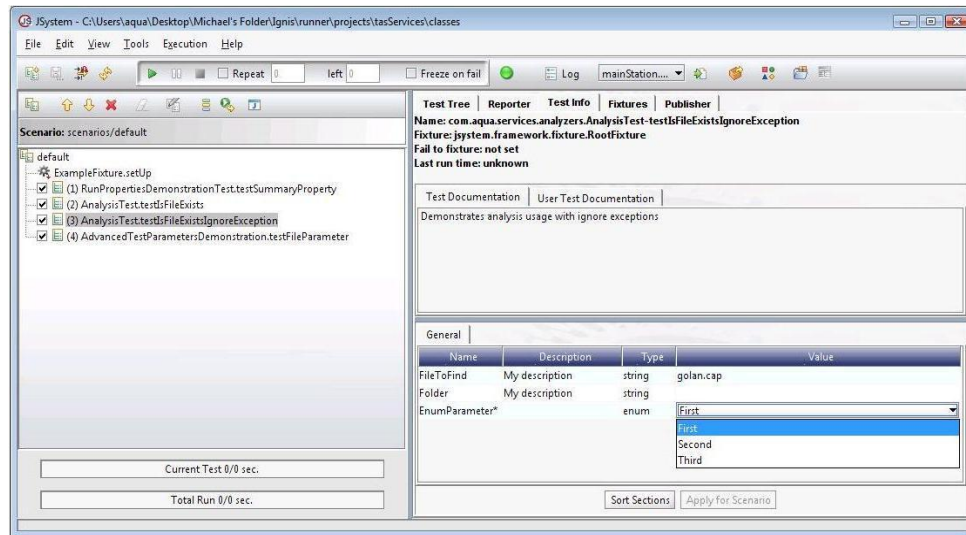


Figure 22: Test Parameter Types HTML Example

The **"PacketSize"** parameter value is also visualized as a drop down list, the list contains the values of the **"PacketSize enum"** class.

4.10.2.5 Supported Objects

In addition to primitives and lists, JSystem supports "**java.io.File**" types and "**java.util.Date**" objects.

4.10.2.6 Supported Objects Code Example

The code example shows a test with test parameters of type "**java.io.File**" and "**java.util.Date**"

```
01 package com.aqua.services.multiuser;
02
03 import java.io.File;
04 import java.util.Date;
05 import junit.framework.TestCase;
06 /**
07  */
08 public class TestAdvancedParametersExample extends SystemTest
Case {
09     private File folder;
10     private Date createTime;
11     public void testVerifyFolderCreateTime() throws Exception{
12         report.report("Test ping");
13     }
14     public File getFolder() {
15         return folder;
16     }
17     public void setFolder(File folder) {
18         this.folder = folder;
19     }
20     public Date getCreateTime() {
21         return createTime;
22     }
23     public void setCreateTime(Date createTime) {
24         this.createTime = createTime;
25     }
26 }
```

Table 35: Supported Objects Code Example

4.10.2.6.1 JSystem JRunner GUI Representation

When editing File parameter values a button appears on the right of the value list called the **"Parameter File Browser"** button.

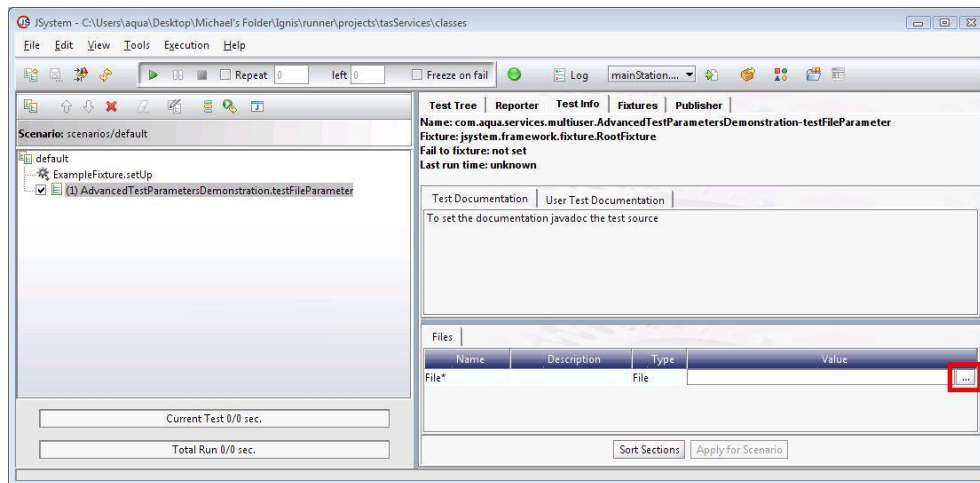


Figure 23: Supported Objects HTML Example

By pressing the value **"Parameter File Browser"** in the **"Test Info"** tab a file browser is opened.

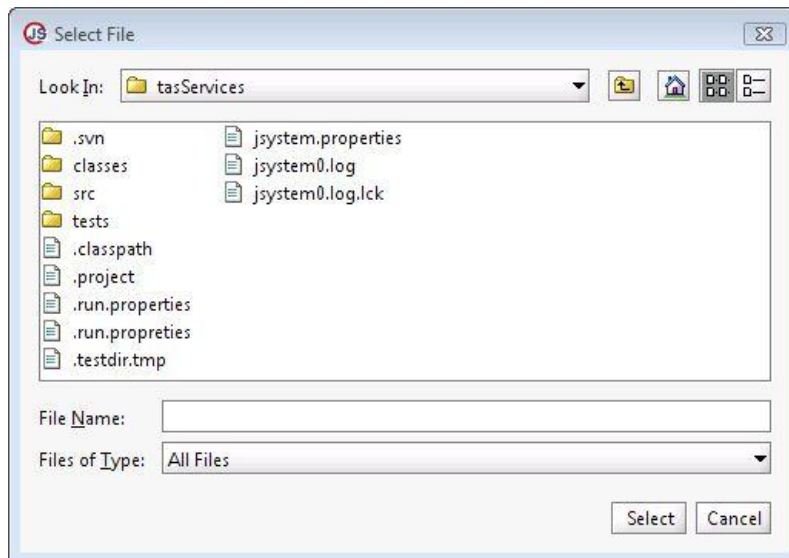


Figure 24: Supported Objects HTML Select File Option

When editing “**CreateTime**” parameter values the “**Parameter File Browser**” button appears on the right of the value list.

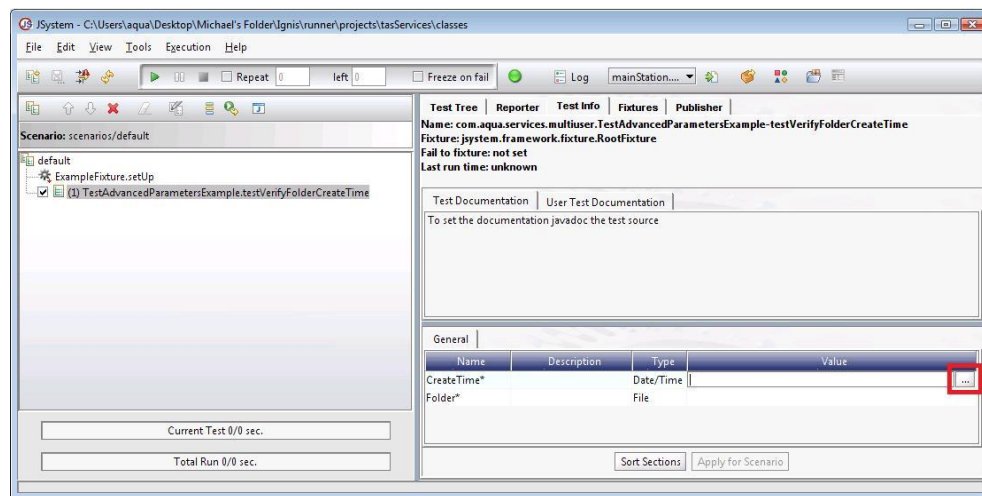


Figure 25: Create Time HTML Example

In order to open the “**Date Editor**”, press the “**Parameter File Browser**” button.

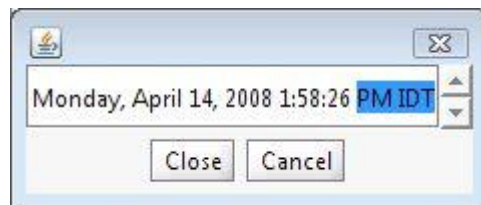


Figure 26: Sample Date Editor Window

4.10.3 Grouping Parameters in Tabs

As the previous example shows, by default test parameters are shown in the “**General**” tab. When there are a large number of parameters, displaying all of the parameters in one tab becomes cumbersome. The “**@section**” annotation, allows the test author to group test parameters in multiple tabs.

In order to group parameters together open the “**javadoc**” of the setter method parameters and add the “**@section sectionName**” annotation.

4.10.3.1 Grouping Parameters in Tabs Code Example

The code below shows a test in which test parameters are grouped into two tabs “**Ping Configuration**” and “**Ping Destination**”.

```
01 package com.aqua.services.multiuser;
02
03 import junit.framework.SystemTestCase;
04 /**
05  * Test parameters simple example.
06  * @author goland
07  */
08 public class TestParametersWithSectionExample extends SystemTe
stCase {
09     private String pingDestination;
10     private int packetSize;
11     public void testPing() throws Exception{
12         report.report("Test ping");
13     }
14     public String getPingDestination() {
15         return pingDestination;
16     }
17     /**
18      * Destination of ping operation.
19      * @section Ping Destination
20      */
21     public void setPingDestination(String pingDestination) {
22         this.pingDestination = pingDestination;
23     }
24     public int getPacketSize() {
25         return packetSize;
26     }
27     /**
28      * Size of packet which will be sent in ping message.
29      * @section Ping Configuration
30      */
31     public void setPacketSize(int packetSize) {
32         this.packetSize = packetSize;
33     }
34 }
```

Table 36: Grouping Parameters in Tabs Code Example

4.10.3.2 JSystem JRunner GUI Representation

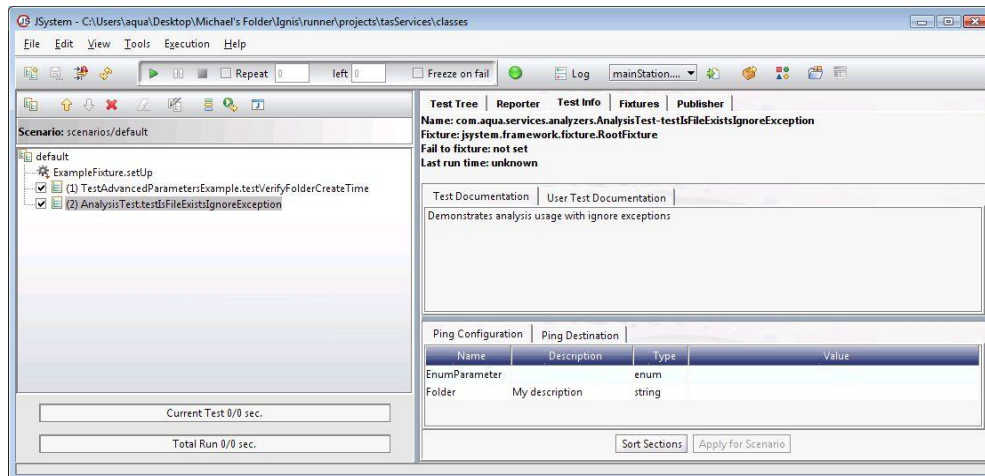


Figure 27: Grouping Parameters in Tabs HTML Example

As can be seen, the “**General**” tab was replaced with two new tabs:

- The “**Ping Configuration**” tab that contains the PacketSize parameter.
- The “**Ping Destination**” tab that contains the Ping Destination parameter.

4.10.3.3 Ordering Tabs

In order to control the order of the tabs in the parameters panel add the method:

4.10.3.4 Ordering Tabs Code Example

The code shows an example implementation of the method “**sectionOrder**” which signals the JRunner the order of parameters tabs.

```
01 public String[] sectionOrder() {  
02     return new String[]{"Tab1", "Tab2"};  
03 }
```

Table 37: Ordering Tabs Code Example

4.10.4 Test Annotations

Tests annotations provide the JRunner additional information about how to present test metadata.

4.10.4.1 Filtering and Ordering Parameters

A typical test class includes a series of test methods; each method uses a subset of test class members. In order for the test to signal the JRunner, the required parameter for the test is the **"@params.include annotation"** method.

4.10.4.2 Filtering and Ordering Code Example

```
01 package com.aqua.services.multiuser;
02 import java.io.File;
03 import junit.framework.TestCase;
04 /**
05  * Params.include example * @author goland
06  */
07 public class TestParamsIncludeExample extends TestCase {
08     private String pingDestination;
09     private int packetSize;
10     private File folder;
11     /**
12      * @params.include pingDestination,packetSize
13      */
14     public void testPing() throws Exception{
15         report.report("Test ping");
16     }
17     /**
18      * @params.include folder
19      */
20     public void testCompareFolder() throws Exception{
21         report.report("Test compare folder");
22     }
23     public String getPingDestination() {
24         return pingDestination;
25     }
26     /**
27      * Destination of ping operation.
28      */
29     public void setPingDestination(String pingDestination) {
30         this.pingDestination = pingDestination;
31     }
32     public int getPacketSize() {
33         return packetSize;
34     }
35     /**
36      * Size of packet which will be sent in ping message.
37      */
38     public void setPacketSize(int packetSize) {
39         this.packetSize = packetSize;
40     }
41     public File getFolder() {
42         return folder;
```

```

43     }
44     public void setFolder(File folder) {
45         this.folder = folder;
46     }
47 }

```

Table 38: Filtering and Ordering Code Example

4.10.4.3 JRunner Filtering and Ordering Example

Under the “**General**” tab, the two items in the “**Name**” column that appear are the, “**Ping Destination**” and “**Packet Size**” string items that were “**included**” in the previous code example.

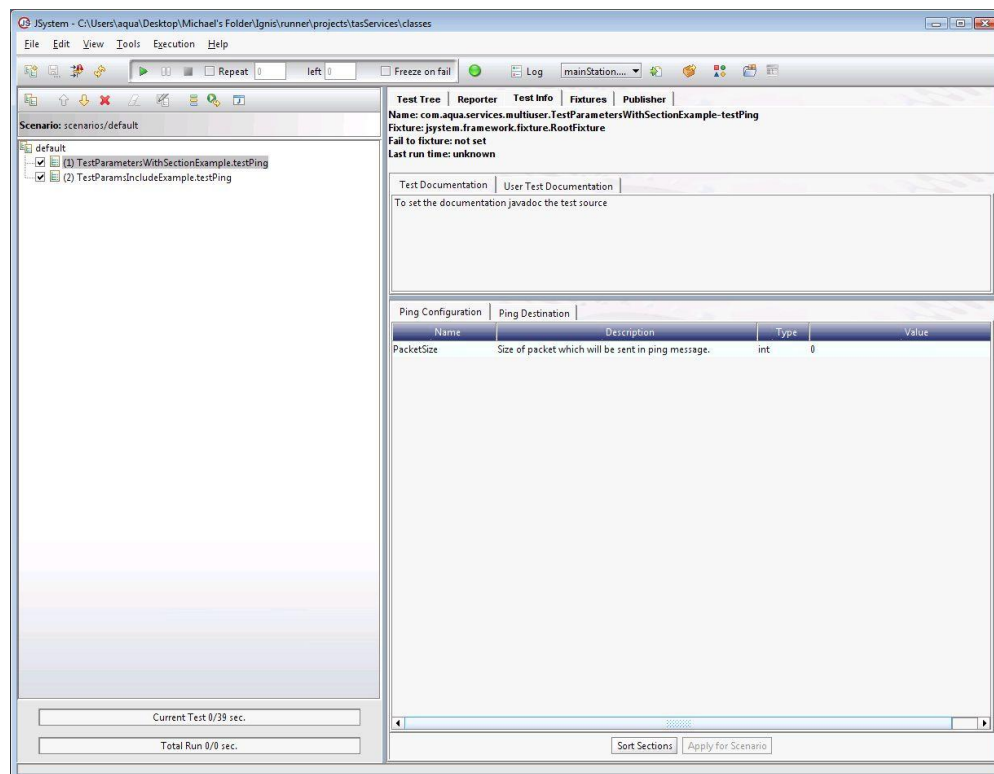


Figure 28: Filtering and Ordering HTML Example

Note: The parameter folder does not appear in the test parameters list.

4.10.4.4 Compare Folder Test in JRunner

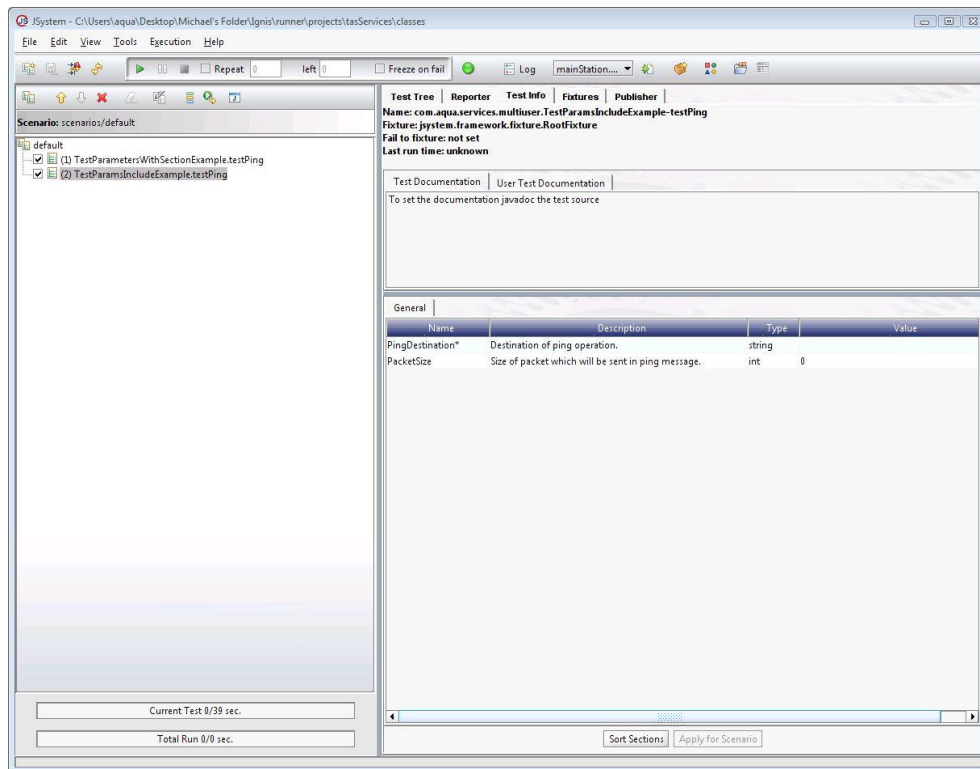


Figure 29: Compare Folder Test in JRunner

Note: In the above screen shot, only the "parameter" folder appears in the list of test parameters.

Instead of the "@params.include" annotation the test author can use the "@params.exclude" annotation that instructs the JRunner which parameters to exclude from parameters list.

Note: If both annotations are used the "params.exclude" is ignored.

4.10.4.5 Test Description

As illustrated in the previous examples the tests appear in the scenario with the “**class name**” and “**method name**”.

In certain cases there is a requirement for the scenario to be more descriptive. This can be achieved by implementing the “**@TestProperties**” annotation.

4.10.4.5.1 Code Example

```
01 package com.aqua.services.multiuser;
02 import jsystem.framework.TestProperties;
03 import junit.framework.SystemTestCase;
04 /**
05  * TestProperties annotation example.
06  * @author goland
07  */
08 public class TestPropertiesAnnotationExample extends SystemTes
tCase {
09     private String pingDestination;
10     private int packetSize;
11     /**
12      * @params.include pingDestination,packetSize
13      */
14     @TestProperties(name="Pings to destination ${pingDestination
}")
15     public void testPing() throws Exception{
16         report.report("Test ping");
17     }
18     public String getPingDestination() {
19         return pingDestination;
20     }
21     /**
22      * Destination of ping operation.
23      */
24     public void setPingDestination(String pingDestination) {
25         this.pingDestination = pingDestination;
26     }
27     public int getPacketSize() {
28         return packetSize;
29     }
30     /**
31      * Size of packet which will be sent in ping message.
32      */
33     public void setPacketSize(int packetSize) {
34         this.packetSize = packetSize;
35     }
36 }
37 }
```

Figure 30: Test Properties Annotation Code Example

4.10.4.6 JRunner GUI Representation

The scenario panel on the left shows the scenario tree, the test does not appear with its **"class name"** and **"method name"**, the description of the test appears as written by the test author.

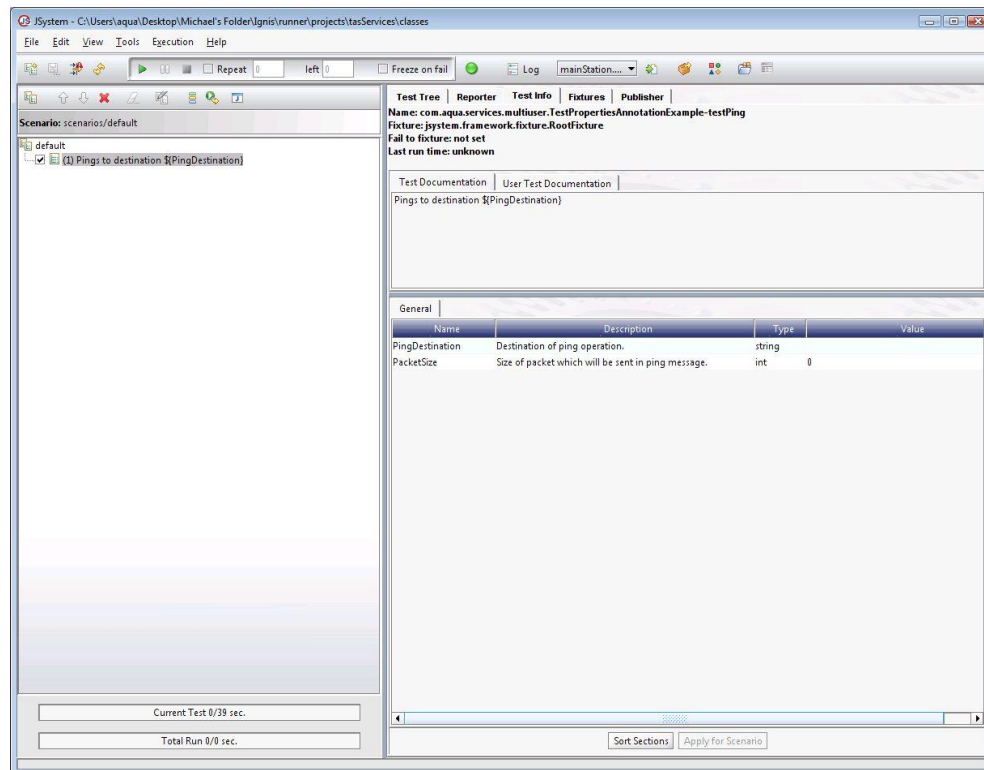


Figure 31: Scenario Tree in the JRunner

Note: If the user changes the ping destination IP and refreshes the JRunner, test description in the scenario will also refresh itself.

4.10.5 Advanced Test Parameter Customization

The test author can manipulate the parameters panel dynamically. The dynamic controls include:

- Defining dependencies between parameters.
- Changing the parameter descriptions according to their value.
- Changing tab grouping dynamically.

These dynamic controls are achieved by implementing the public void `handleUIEvent` "**HashMap<String,Parameter> map,String methodName**" method.

The "**handleUIEvent**" method is called by the framework whenever the user makes a change to one of the parameters.

This method receives the "**HashMap<String,Parameter> map**" of the test method parameters the key is the parameter name as it appears in the parameters panel and the value is a "**Parameter**" object that represents the parameter.

By operating on the parameter instance the programmer can enable and or disable parameters, change its description, tab grouping, type and value.

4.10.5.1 Test Parameters Customization Code Example

The code is an example of the Implementation of the "**handleUIEvent**" method.

```
01 package com.aqua.services.multiuser;
02 import java.io.File;
03 import java.util.HashMap;
04 import jsystem.framework.TestProperties;
05 import jsystem.framework.scenario.Parameter;
06 import junit.framework.SystemTestCase;
07 public class DynamicUpdateDemonstration extends SystemTestCase
08 {
09     private File file;
10     private File file2;
11     private boolean recursive = true;
12     /**
13      * Compares the content of files <code>file1</code>,<code>file2</code>.
14      * If file and file2 are folders compares the content of the
15      * files in the folders.
16      * Additional folder comparison parameters:
17      * recursive - whether to do a recursive compare.
18      * between the folders
19      */
20     @TestProperties(name="Compares the content of folders ${file}
21     },${file2}")
22     public void testCompareFiles() throws Exception{
23     }
24     /**
25      */
26     public void handleUIEvent(HashMap<String,Parameter> map,String
27     methodName) throws Exception {
28         Parameter fileParameter = map.get("File");
29         Parameter file2Parameter = map.get("File2");
30         Parameter recursiveParameter =map.get("Recursive");
31         if (fileParameter.getValue() == null){
```

```

28     fileParameter.setValue("");
29 }
30 if (file2Parameter.getValue() == null){
31     file2Parameter.setValue("");
32 }
33 File fileObject = new File(fileParameter.getValue().toString());
34 File file2Object = new File(file2Parameter.getValue().toString());
35 /**
36  *
37  */
38 if (fileObject.isDirectory()){
39     fileParameter.setDescription("Source folder");
40 }else {
41     fileParameter.setDescription("Source file");
42 }
43
44 if (file2Object.isDirectory()){
45     file2Parameter.setDescription("Source folder");
46 }else {
47     file2Parameter.setDescription("Source file");
48 }
49 /**
50  *
51  */
52 if (file2Object.exists() && file2Object.isDirectory() && fileObject.exists() && fileObject.isDirectory()){
53     recursiveParameter.setEditable(true);
54 }else {
55     recursiveParameter.setEditable(false);
56 }
57 }
58 public File getFile() {
59     return file;
60 }
61 public void setFile(File file) {
62     this.file = file;
63 }
64 public File getFile2() {
65     return file2;
66 }
67 public void setFile2(File file2) {
68     this.file2 = file2;
69 }
70 public boolean isRecursive() {
71     return recursive;
72 }
73 public void setRecursive(boolean recursive) {
74     this.recursive = recursive;
75 }
76 }

```

Table 39: Advanced Test Parameters Customization

4.10.5.2 DynamicUpdateDemonstration Test Class

The “**DynamicUpdateDemonstration**” class contains one test method: the “**testComapreFiles**” this test case gets two parameters, file and file2 as they appear in the previous code example in lines “08” and “09”. The description of each file parameter that points to a directory is the “**Source folder**”, if it points to a file or to a non existing file, the description will be “**Source file**”. If both file parameters point to directories, the recursive parameter is enabled, otherwise it is disabled.

4.10.5.3 Understanding the “handleUIEvent” Method Implementation

The following code examples helps explain the implementation of the “**handleUIEvent**” method, as it appears in the previous code example.

The code demonstrates the map retrieval function of the “**File**” and “**File 2**” Parameter instance representing each test parameter.

```
1 Parameter fileParameter = map.get("File");
2     Parameter file2Parameter = map.get("File2");
3     Parameter recursiveParameter =map.get("Recursive");
```

Table 40: handleUIEvent Code Example

If the value of files is “**null**” the code automatically replaces it with empty string.

```
1 if (fileParameter.getValue() == null){
2     fileParameter.setValue("");
3 }
4 if (file2Parameter.getValue() == null){
5     file2Parameter.setValue("");
6 }
```

Table 41: getValue Code Example

If the file parameter points to a directory set description such as “**Source folder**” if the directory is not present or it does not exist it is set it to “**Source file**”

```
01 if (fileObject.isDirectory()){
02     fileParameter.setDescription("Source folder");
03 }else {
04     fileParameter.setDescription("Source file");
05 }
06 if (file2Object.isDirectory()){
07     file2Parameter.setDescription("Source folder");
08 }else {
09     file2Parameter.setDescription("Source file");
10 }
11 }
```

Table 42: null Replace Code Example

If both the file parameters point to a directory then the code sets the recursive parameter to be editable if not it is disabled.

```
1 if (file2Object.exists() && file2Object.isDirectory() && fileObject.exists() && fileObject.isDirectory()) {
2     recursiveParameter.setEditable(true);
3 } else {
4     recursiveParameter.setEditable(false);
5 }
```

Table 43: Point to Directory Code Example

4.10.6 Test and Parameters Annotations

JSystem provides an extended “**TestProperties**” annotation called the “**ParameterProperties**” annotation. This annotation replaces the old “**javadoc**” annotations used until the release of JSystem 5.1, such as “**params.include**”, “**params.exclude**” and section.

4.10.6.1 Annotations Usage

An annotation should be written just above the method it relates to, if there is also a “**javadoc**” for the method it should be between them.

For example,

```
1 /**
2  * Test the parameters feature
3  *
4  */
5 @TestProperties(name = "check that one parameter is visible")
6 public void testOneParamAndGibrish() {
7     report.report("report something");
8 }
```

Table 44: Annotations Code Example

Note: all annotation fields can be written in any order, comma separated. ■

4.10.6.1.1 Code Example

```
1 @TestProperties(name = "check that one parameter is visible",
2                 paramsInclude={"intValue"})
3 public void testOneParamAndGibrish(){
4     report.report("report something");
5 }
```

Table 45: Annotations Code Example 2

4.10.7 @Test Properties Annotation

TestProperties annotation is added before a test method and supports the following fields.

1. **name** – a meaningful test name, displayed on the Scenario tree and in the html logs. String.
2. **Group** - used for filtering tests in the JRunner and String array.
3. **returnParam** – used to define return parameters used by the flow control and String array.
4. **paramsInclude** – defines which test parameters are visible in the JRunner GUI for the test method. (An empty string means "**reveal no parameters**"). String array.
5. **paramsExclude** - defines which test parameters are not visible in the JRunner GUI for the test method. (An empty string means "**reveal all parameters**"). String array.

Note: "paramsInclude" has priority and overrides "paramsExclude".

Note: The correct syntax for an empty parameters list is "paramsInclude={}" and not "paramsInclude={}". The same is true for "paramsExclude".

4.10.7.1 @TestProperties Annotation Code Example

The following code example refers to test property items four and five in the previous section called “**Test Properties Annotations**”.

The following code excerpt is an example of a method in earlier JSystem versions.

```
1 /**
2  * Test the parameters feature
3  * @params.include intParam stringParam
4  */
5 public void testEmptyInclude() {
6     report.report("intParam = "+intParam);
7 }
```

Table 46: Old Code Example

The following code excerpt is an example of a method in JSystem version 5.1.

```
1 /**
2  * Test the parameters feature
3  */
4     @TestProperties(paramsInclude = {"intParam","stringParam"})
5 public void testEmptyInclude() {
6     report.report("intParam = "+intParam);
7 }
```

Table 47: New JSystem 5.1 Method Code Example

4.10.8 @ParameterProperties Annotation

The “**ParameterProperties**” annotation is added before a parameter setter and supports the following fields:

- **section** – defines the tab that the parameter belongs in the JRunner parameters - panel. String
- **description** – a short description for the parameter, appears in the parameters - panel

4.10.8.1 @ParameterProperties Annotation Code Example

Setter code example as it appears in earlier JSystem versions.

```
1 /**
2  * the user name for the DB
3  * @section MySection
4  */
5 public void setStringParam(String stringParam) {
6     this.stringParam = stringParam;
7 }
```

Table 48: Parameter Properties Old Code Example

Parameter properties code example in JSystem version 5.1.

```
1 @ParameterProperties(section="MySection",
2                     description=" the user name for the DB")
3 public void setStringParam(String stringParam) {
4     this.stringParam = stringParam;
5 }
```

Table 49: New Parameters Property Code Example

4.10.9 System Object Lifecycle

In addition to instantiating a system object the framework can also manages the system object lifecycle.

4.10.10 Initialization and Termination

When a system object is fetched for the first time, it is entered into the internal system map. The next time that the system object is requested there is no need to re-initiate the system object; it is simply fetched from the internal system map.

System objects run in one of two life time modes:

1. Test life time
2. Permanent life time.

When running in "**test life time**" mode, the system object is disposed at the end of each test. As a result, the "**close()**" method is called at the end of the test. The next time the system object is requested using the "**getSystemObject**" method; the system object is instantiated again. When running in "**permanent life time**" mode system object it is disposed of before the test JVM is disposed.

The default system object life time mode is "**permanent lifetime**".

Reference: For more detailed information about JRunner run mode, read the Chapter Error! Reference source not found. on page Error! Bookmark not defined.

4.10.11 Pause and Stop

The JRunner allows the user who executes the scenario to pause and stop scenario execution.

Before execution of scenario is paused the "**pause**" method of all system objects in framework map is called.

When execution is resumed, the "**resume**" method is invoked.

Note: When stopping a tests execution by using the  "stop" button, the scenario execution is stopped without first disposing the system objects.

4.10.12 TestListener Class

The system object can implement the interface "**junit.framework.TestListener**" method. By doing so, the system object will be signaled upon test execution of events.

An example of a system object that implements the "**TestListener**" interface is as follows:

The "**addError**", "**addFailure,endTest**" and "**startTest**" are "**TestListener**" methods. The "**startTest**" and "**endTest**" are invoked whenever a test starts and or ends the "**addError**" is called whenever a test fails due to execution error; the "**addFailure**" method is called whenever a test fails due to assertion problem.

4.10.12.1 TestListener Class Code Example

Below is a code

```

01 package com.aqua.services.systemobject;

03 import jsystem.framework.system.SystemObjectImpl;
04 import jsystem.utils.FileUtils;
05 import junit.framework.AssertionFailedError;
06 import junit.framework.Test;
07 import junit.framework.TestListener;
08
09 public class HelloWorldWithTestListener extends SystemObjectImpl
implements TestListener{
11     private String message;
12     private String fileName;
13     public void init() throws Exception {
14         super.init();
15         report.report("Hello world init");
16     }
17
18     public void close(){
19         report.report("Hello world close");
20         super.close();
21     }
22
23     public void getHelloMessage() throws Exception {
24         report.report("Hello Message",getMessage(),true);
25     }
26
27     public void readFromFile() throws Exception {
28         String textFromFile = FileUtils.read(getFileName());
29         setTestAgainstObject(textFromFile);
30     }
31
32     public void writeToFile(String text) throws Exception {
33     }
34
35     public String getMessage() {
36         return message;
37     }
38
39     public void setMessage(String message) {
40         this.message = message;
41     }
42
43     public String getFileName() {
44         return fileName;
45     }
46
47     public void setFileName(String fileName) {
48         this.fileName = fileName;
49     }
50
51     @Override
52     public void addError(Test arg0, Throwable arg1) {
53         // Will be called
54     }
55
56     @Override
57     public void addFailure(Test arg0, AssertionFailedError arg1) {
58         // TODO Auto-generated method stub
59     }
60
61
62     @Override
63     public void endTest(Test arg0) {
64         // TODO Auto-generated method stub
65     }
66
67
68     @Override
69     public void startTest(Test arg0) {
70         // TODO Auto-generated method stub
71     }
72
73 }
74 }

```

4.11 Additional Services

The JSystem Automation Platform contains additional smaller services that are used for a wide range of testing purposes. They include run properties and summary properties.

4.11.1 RunProperties

Run properties are required in order to enable tests to communicate with each other during a JSystem run. These run properties exist in the context of the test run and are automatically cleared when the test run finishes. In a typical operating setup of the RunProperties service, one test can set a property and another test can read the property and verify its value simultaneously.

4.11.1.1 Setting the Run Property Code Example

```
01    * Run properties are properties which
02    * purpose is to help tests to share information.
03    * When running in run.mode 1 (1 jvm for all tests) , this is
    not a problem since
04    * tests can share data in JVM's memory, but when running in
    run.mode 2,4
05    * (jvm for each test and jvm for each internal scenario), te
    sts can't
06    * share data in JVM's memory, this is when the run propertie
    s can be used.
07    */
08    public void testRunProperties() throws Exception {
09        RunProperties.getInstance().setRunProperty("myProp", "val1"
10    );
11    }
```

Table 50: Additional Services Code Example

4.11.1.2 Reading Property Value Code Example

```
2    * When running this test after {@link #testRunProperties()},
3    * the value of val will be 'val1'
4    */
5    public void testRunPropertiesSecondTest() throws Exception {
6        String val = RunProperties.getInstance().getRunProperty("myP
    rop");
7        assertEquals(val, "val1");
8    }
```

Table 51: Reading Properties Code Example

4.11.2 Summary Properties

Summary properties are run properties that are not deleted between test runs. These summary properties are set by tests and not read by other tests, they are used by two other JSystem services:

- The HTML JReporter
- The JRunner publisher

The summary properties service has four pre defined properties that are created by the framework:

1. **Date** – Scenario execution start date and time.
2. **Station** – Machine IP.
3. **User** – User name taken from local machine.
4. **Version** – Version of a tested product. The default setting is set to “**unknown**”.

4.11.2.1 Summary Properties Code Example

The values of all built-in summary properties can be overridden by the programmer. In order to set the summary properties write the following code:

```
1  /**      */
3  public void testSummaryProperty() throws Exception {
4      Summary.getInstance().setProperty("permanentProperty", "pro
pValue");
5  }en
```

Table 52: Summary Property Code Example

4.11.3 Summary Properties in the HTML Reporter

All summary properties are added to the main page of the HTML JReport.

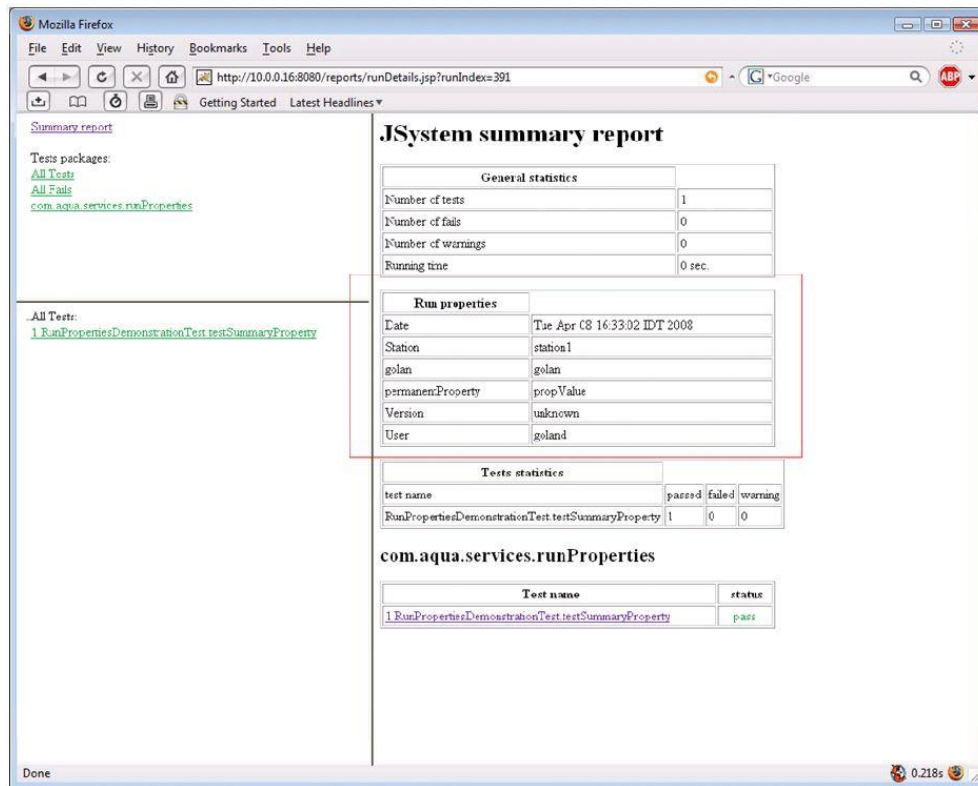


Figure 32: Summary Properties in the HTML Reporter

4.11.3.1 Summary Properties in the Publishing Process

When publishing test results via the publish dialog or through publish event the values of the system summary properties appears in the publish dialog.

For a more in depth understanding of a "Publish Event" see Error! Reference source not found. on page Error! Bookmark not defined..

4.12 Additional Language Support

The JSystem framework includes infrastructure for developing plug-ins for scripting languages (see ...). Using this infrastructure we have created a plug-in for running Jython tests.

4.12.1 Jython Test Support

Jython (www.jython.org) is a Java implementation of the Python scripting language (www.python.org). Its concise syntax and dynamic nature makes it a perfect fit for writing test cases and the latest version of JSystem allows the test author to use it to write JSystem system tests.

A test in Jython is written as a class in a ***.py** file using Jython's **"unittest"**, a library for running unit tests in Jython and Python, similar to JUnit for Java.

The following example test is written in Jython using an imaginary system object that can check for the existence of a file in a directory:

```
01 import unittest
02 from jyutils import *
03 class FileTests(SystemTestCase):
04     __parameters__ = dict(
05         directory = Parameter(),
06         file = Parameter()
07     )
08     def setUp(self):
09         self.sysobj = system.getSystemObject("fileChecker")
10     def test1(self):
11         report.report("Directory : %s" % self.directory)
12         report.report("File : %s" % self.file)
13         self.assertTrue(self.sysobj.isFileInDir(self.file, self.directo
ry))
```

Table 53: Jython Code Example

4.12.1.1 Writing Jython Tests

Each Jython test file should start with the **jyutils import *** form line, in order to allow the tests access to various JSystem features. Each file should contain one or more class, each extending the **"SystemTestCase"**. Similar to JUnit (version 3) the tests are described in methods whose names start with the word **test**, with **"setUp"** called before each test method is run and a **"tearDown"** method is called after each test.

The user can define parameters for the test that will be displayed in the JRunner user interface by creating a **__parameters__** static data member, which is a **"dict"** of parameter names mapped to the **"Parameter"** objects. The parameter values are accessible as data members in the test methods themselves (see references to **"self.file"** and **"self.directory"** above).

Jython test files should be put in a subdirectory of the current project path alongside the **"*.class"** files of the Java tests.

The user can create scenarios that mix Jython test cases and regular JUnit test cases. In order to use the new Jython capabilities, add the following line to the **"jsystem.properties"** file.

```
1 script.engines=jsystem.framework.scripts.jython.JythonScriptEng  
ine
```

Table 54: JSystem Properties File