LISA Developer's Guide

The **LISA Developer's Guide** contains instruction for the more technical user of LISA. LISA functionality can be extended with programmers as there is a high amount of customization available within LISA. This software used to be known as the LEK (LISA Extension Kit) and is now also being referred to as the SDK (Software Development Kit).

The following topics are available in this guide.

Also available is the jdbridge user guide:

LISA Java .NET Bridge

The LISA Integration API

The LISA software provides an integration Application Programmer's Interface (API) that allows you to manage LISA test execution within Java-based server-side components. This chapter details the basic concepts central to the LISA integration API and includes the following sections:

- LISA Integration API Concepts
- LISA Integration Flow
- The LISA Integration Process

⚠ The LISA software currently supports only Java-based server side components. See the ITKO website, www.itko.com, for future releases supporting other types of server-side applications.

LISA Integration API Concepts

The LISA software provides a powerful framework for testing server-side components. However, if a test fails, the framework provides little information on why the test failed. In addition, the component under test often calculates information that is of interest to the tester, but is difficult to retrieve.

The LISA integration API provides several elements for getting information out of server-side components. These elements include:

- **Integrators**: Application-specific Java classes that coordinate communication with the running test. For example, an EJB Integrator informs an EJB component whether LISA integration is turned on. For more information about integrators, see Integrating Components.
- **TransInfo Class**: Abbreviation for "transaction information," this class is used by the integrated component to indicate testing events, such as test failure and success. For more information about using the TransInfo class, see Integrating Components.
- **HasLisaIntegrator Interface**: Implemented by object types returned by methods integrated using Java-based integrators. This provides the running test with the information it needs about how the test has changed while testing the component. For more information about implementing the HasLisaIntegrator interface, see Constructing a Response Object.
- **Integration Filters**: Application-specific LISA filters that coordinate communication with the integrated component. For example, the Servlet Filter turns on integration support for a servlet component and specifies a node to execute if the servlet returns specific values. For more information about integration filters, see LISA Integration Filters.
- **Integration Assertions**: Special assertion elements that take advantage of the additional information provided by integrated server-side components. For example, a Check LISA Integrator Response Assertion can set the next test step based on the build status reported by

the integrated component. For more information about integration assertions, see LISA Integration Assertions.

LISA Integration Flow

1. The test case developer adds an application-specific integration filter to a test case. When the test is run, the filter turns on integration support for the type of application indicated. For example, the test case developer adds a Servlet Filter to test an integrated servlet.
2. When the server is invoked, test-enabled server components use the application-specific integrator class and the TransInfo class to establish meaningful communication with the LISA test case that is running. For example, if a servlet fails in some way, it can call **TransInfo.setBuildStatus()** to note the failure to LISA. For more information about the TransInfo class, see the LISA Javadoc.
3. The filter automatically processes responses from the system under test. The filter logs important information and processes any commands from the tested component.

The LISA Integration Process

To integrate to the LISA integration API

1. Make sure **lisaint2.0.jar** is in your classpath. You can find **lisaint2.0.jar** in the **LISA_HOME/lib** directory.
2. Add LISA integration to the method. For more information about integrating LISA into a server-side component, see Integrating Server-Side Components.
3. Handle the output from the integrated method. For more information about handling integrated output, see Handling Integrated Output.
4. Use integration filters in the test case that tests the server-side component. For more information about integration filters, see LISA Integration Filters.
5. Use integration assertions in the test case that tests the server-side component. For more information about integration Assertions, see LISA Integration Assertions.

Integrating Components

This chapter explains how to integrate server-side components using the LISA integration API.
Topics include:

- Integrating Server-Side Components
- Collecting Transaction Information
- Integrators
- Handling Integrated Output

Integrating Server-Side Components

To LISA-enable a server-side component, complete the following steps:

LISA-enable a Server-Side Component

1. Import the necessary classes. The component must at least import the appropriate integrator and the **TransInfo** class. For example, to import the classes necessary to integrate a servlet, add the following import statements:

```
import com.itko.lisaint.servlet.ServletIntegrator;
import com.itko.lisaint.TransInfo;
```

For more information about the TransInfo class, see the LISA Javadoc.

2. Declare local variables to hold state. The integrated component needs at least an integrator and a TransInfo variable, as seen in the following code:

```
ServletIntegrator si = null;
TransInfo ti = null;
```

3. Start a transaction. The LISA integration API lets you control when the integrated component begins to communicate with the LISA software. All the communication needs to happen in context of a transaction. Therefore, before beginning to communicate, start a transaction using the application-specific integrator. You can make sure that LISA is on before that by checking with the application-specific integrator class. The following code checks if LISA is on, then retrieves the servlet integrator from the integrator class and stores it in the variable **si**. The code then starts the transaction using the servlet integrator and stores the result in the TransInfo object.

```
if( ServletIntegrator.isLisaOn( request ) ) {
    si = ServletIntegrator.getServletIntegrator( request );
    ti = si.startTransaction( "Hello World" );
}
```

4. Interact with the test case.

- Report component status. The TransInfo class provides a method, **setBuildStatus()**, that allows you to specify information that LISA can use to determine how to run the test. For example, to tell LISA that the component has failed, set the build status to STATUS_FAILED, as seen in the following code:

```
if( /* component failed */ ) {
    // ...
    ti.setBuildStatus( TransInfo.STATUS_FAILED, failMsg );
}
```

For more information about valid build status codes, see Build Status.

- Set LISA properties. The TransInfo class provides a method, **setLISAProp()**, that allows you to set an arbitrary LISA property. This is a very useful mechanism for getting information out of an integrated component, especially in conjunction with the Ensure Property Matches Expression assertion.

For example, if your application calculates the Dow average, you could store that value in a LISA property that can be checked while testing. The following code creates a new LISA property called DOW_AVERAGE and assigns the value CalculatedDowAverage.

```
ti.setLISAProp( "DOW_AVERAGE", CalculatedDowAverage );
```

This mechanism can be used to send data cache data from the system under test to the Test Manager as LISA property values. These property values can be any serializable object, including strings, any of the Java object wrappers for primitive types, or even your own classes as long as the LISA class path contains the proper classes to deserialize the data.

- Make assertions. The TransInfo class provides several methods that make assertions, then take some action if the assertion fails. For example, the **assertFailTest()** method takes a Boolean value and makes an assertion that the value is true. If the value is false, the method instructs LISA to fail the test, passing the string message that is logged.

```
ti.assertFailTest( boolean_assertion, "ASSERT FAILED" );
```

For more information about TransInfo assertion methods, see Assertion Methods.

- Force the next node. On rare occasions, it is very difficult to get information from the server-side component back to the test case. In this situation, it is helpful to be able to force the test case to visit a specific test step if the condition arises. The TransInfo class provides a method, **setForcedNode()**, that overrides the next test step as determined by the Integrating Components test case. In the following code, the developer forces LISA to make specialStep the next test step if the special_condition is true.

```
if (special_condition)
ti.setForcedNode("specialStep");
```

5. Send the response back to LISA. For more information about sending responses back to LISA, see Handling Integrated Output.

JSP Tag Library

In addition to the Java API shown previously, LISA provides a JSP tag library for Java web development. This API makes using the Integration API easy for JSP developers.

Pure XML Integration

ITKO makes available a pure XML form of the Integration for web applications on an as-needed basis. If you require this form of integration, contact your sales or support representative.

Test Components

Sometimes a server-side application has a number of checks and it is not clear where the test step actually failed. You can separate out parts of the transaction using sub-components and report their individual statuses. The **com.itko.lisaint.CompInfo** class represents a sub-component.

For example, the following code creates a new sub-component of the transaction and sets its build status to STATUS_SUCCESS:

```
CompInfo ci = ti.newChildComp( "SUBCOMP" );
ci.setBuildStatus( CompInfo.STATUS_SUCCESS, "" );
ci.finished();
```

Sub-components are treated as part of the overall transaction. If one sub-component fails, the entire transaction fails. The following code creates a sub-component and sets the build status to STATUS_FAILED. In this case, the entire transaction fails.

```
CompInfo ci = ti.newChildComp( "SUBCOMP" );
ci.setBuildStatus( CompInfo.STATUS_FAILED, "" );
```

For more information about the CompInfo class, see the LISA Javadoc.

Collecting Transaction Information

The TransInfo class encapsulates all the information gathered about the execution of a specific "transaction" of the system under test, and informs the running test case of the status of this transaction. The LISA instance requesting this transaction works with this object. A transaction is a round-trip from a test instance to the system under test and back. Transactions can be broken up into sub-components and reported at that level with the **CompInfo** object.

To construct a TransInfo object, call the **startTransaction** method on the appropriate integrator. For example, the following code creates a TransInfo in a servlet:

```
TransInfo ti = si.startTransaction( "Hello World" );
```

For more information about the TransInfo class, see the LISA Javadoc.

Build Status

The TransInfo class provides a method, **setBuildStatus**, that lets you specify information that LISA can use to determine how to run the test. The **setBuildStatus** method takes a string constant whose possible values are defined on the TransInfo class. Status constants include:

- **S - STATUS_SUCCESS**: The component executed as expected.
- **U - STATUS_UNKNOWN**: The status is not known.
- **R - STATUS_REDIRECT**: The component issued a redirect (only valid for some systems).
- **I - STATUS_INPUTERROR**: The component failed due to bad inputs.
- **E - STATUS_EXTERNALERROR**: The component failed due to external resource errors.
- **F - STATUS_FAILED**: The component failed, presumably not because of inputs or external resources.

For more information about the TransInfo class, see the LISA Javadoc.

Assertion Methods

The TransInfo class provides several methods that make assertions, and then take some action if the assertion fails. Assertion methods include:

- **assertLog(boolean expr, String msg)**: If the assertion expr is false, the message is written to the log.
- **assertEndTest(boolean expr, String msg)**: If the assertion expr is false, tell LISA to end the test normally, and write the message to the log.
- **assertFailTest(boolean expr, String msg)**: If the assertion expr is false, tell LISA to fail the test, and write the message to the log.
- **assertFailTrans(boolean expr, String msg)**: If the assertion expr is false, tell LISA to consider the transaction failed, and write the message to the log.
- **assertGotoNode(boolean expr, String stepName)**: If the assertion expr is false, tell LISA to execute the stepName test step next. This can also be a property name in double curly brace (x) notation.

For more information about the TransInfo class, see the LISA Javadoc.

Integrators

An integrator is an application-specific Java class that coordinates communication with the running test. For example, an EJB Integrator informs an EJB component whether LISA integration is turned on.

The **com.itko.lisaint.Integrator** abstract class provides the base set of functionality for the following integrators:

- Java Integrator
- EJB Integrator
- Servlet Integrator

Each integrator coordinates with the running test on whether LISA and LISA integration is turned on. After this is determined, the primary responsibility of the integrator is to start a transaction and provide access to the TransInfo object.

For more information about creating a TransInfo object, see Collecting Transaction Information.

For more information about the Integrator class, see the LISA Javadoc.

Handling Integrated Output

In addition to the transaction information, the integrated component must return the results of the component to the LISA software. For servlets, this involves wrapping servlet responses. For Java-based components, this involves constructing a response object.

Wrapping Servlet Responses

The LISA software integrates with web-based applications by embedding a streamed version of the integrator into the HTML output of the web server. The ServletIntegrator object is converted to ASCII text and wrapped in an HTML comment. The ServletIntegrator class provides a method, **report**, that takes an output stream and performs the wrapping described before sending it back to the LISA software.

The following code wraps the servlet response and sends it back to the LISA software, if the ServletIntegrator indicates that LISA is on:

```
if( ServletIntegrator.isLisaOn( request ) )
si.report( out );
```

For more information about the ServletIntegrator class, see the LISA Javadoc.

Constructing a Response Object

The LISA software integrates with Java-based applications by enforcing that either the method return value types or the class you are executing itself implement the **com.itko.lisaint.java.HasLisaIntegrator** interface.

For example, assume you have implemented an object with a LoginInfo object as a return value to the login(String uid, String pwd). The LoginInfo object maintains the information about whether the test succeeded or failed. To test that method, a LISA test case author executes the **login** method, then queries the returned LoginInfo object for details to determine the success or failed state.

To implement the **com.itko.lisaint.java.HasLisaIntegrator** interface, implement the **getLisaIntegrator()** method to provide an XML representation of your integration object to LISA.

Most implementations also provide a **setLisaIntegrator()** method and store the state in a member variable, as in the following code:

```
public class LoginInfo implements HasLisaIntegrator {

    private JavaIntegrator lisa;

    public String getLisaIntegrator() {
        return lisa.toXML();
    }

    public void setLisaIntegrator(JavaIntegrator obj) {
        lisa = obj;
    }

}
```

For more information about the HasLisaIntegrator and LoginInfo classes, see the LISA Javadoc.

Testing Integrated Components

This chapter explains how to test integrated server-side components using LISA integration filters and assertions. Topics include:

- LISA Integration Filters
- LISA Integration Assertions

LISA Integration Filters

An integration filter is an application-specific LISA filter that coordinates communication with the integrated component. The inclusion of an integration filter indicates two things:

- The LISA software should turn on support for integration with that type of server-side component.
- When an integrated server-side component returns results that match the attributes of the filter, execute a specific test step as the next test step.

For example, the Servlet Filter turns on integration support for a servlet component and specifies a test step to execute if the servlet returns specific values. A test case developer typically defines multiple integration filters of a single type. Each filter checks a specific condition and sets the next test step accordingly.

The Model Editor provides built-in support for three integration filter types:

- LISA Integration Filter for Java Applications
- LISA Integration Filter for Servlet Applications
- LISA Integration Filter for EJB Applications

Defining an integration filter in a test case indicates that LISA wants to turn on integration for that type and to listen for responses of that type. To define an integration filter, create the filter, select the type, and set the attributes.

- **Trans Build Status:** The server-side component has set the build status on the TransInfo to a specific value. This field takes one or more build status code letters. For example, to define a filter that fires if the build status is set to any terminating status, enter the value **IEF**.
- **Trans Build Message Expression:** The server-side component has supplied a string as the message parameter to the setBuildStatus method of the TransInfo that matches the specified regular expression.
- **Component Name Match:** The server-side component created a sub-component with a name that matches the value of this field. Possible values for the field include:
    - empty: Do not evaluate, meaning that there is never a match.
    - *: Match anything except a null.
    - anything else: Evaluated as a regular expression.
- **Component Build Status:** The server-side component has set the build status on the CompInfo to a specific value. This field takes one or more build status code letters. For example, to define a filter that fires if the build status is set to any terminating status, enter the value **IEF**.
- **Component Build Message Expression:** The server-side component has supplied a string as the message parameter to the setBuildStatus method of the TransInfo that matches the specified regular expression.
- **Exception Type Match:**The server-side component threw an exception with a type that matches the value of this field. Possible values for the field include:
    - empty: Do not evaluate, meaning that there is never a match.
    - *: Match anything except a null.
    - anything else: Evaluated as a regular expression.
- **Max Build Time (millis):** The time to execute the server-side component took less than the number of milliseconds specified.
- **On Transaction Error Node:** If the server-side component matches the attributes specified, then execute the node specified.
- **Report Component Content:** If this box is selected, the servlet application tested should include the actual content of the components

rendered in the output.

⚠ Only select the Report Component Content box if necessary, as it can be very bandwidth intensive. Also, the check box is only a request. Servlet applications may ignore the request and not provide the component content.

For more information about filters, see the *LISA User Guide*.

LISA Integration Assertions

An integration assertion is an application-specific LISA assertion that executes a specific test case as the next test case if an integrated server-side component returns results that match the attributes of the assertion.

For example, the Check LISA Integrator Response Assertion specifies a node to execute if the server-side component returns specific values. A test case developer typically defines multiple integration assertions of a single type. Each assertion checks a specific condition and sets the next node accordingly.

The Model Editor provides built-in support for three integration assertion types:

- Check LISA Integrator Response
- Check LISA Integrator Component Content Response
- Check LISA Integrator Reporting Missing Data

To define an integration assertion, create the assertion and set the attributes based on the type of assertion, as outlined in the following section.

Check LISA Integrator Response

The Check LISA Integrator Response assertion specifies a test case to execute if the server-side component returns specific values. Attributes include:

- **Trans Build Status:** The server-side component has set the build status on the TransInfo to a specific value. This field takes one or more build status code letters. For example, to define an assertion that executes if the build status is set to any terminating status, enter the value **IEF**.
- **Trans Build Message Expression:** The server-side component has supplied a string as the message parameter to the setBuildStatus method of the TransInfo that matches the specified regular expression.
- **Component Name Match:**The server-side component created a sub-component with a name that matches the value of this field. Possible values for the field include:
    - empty: Do not evaluate, meaning that there is never a match.
    - *: Match anything except a null.
    - anything else: Evaluated as a regular expression.
- **Component Build Status:** the server-side component has set the build status on the CompInfo to a specific value. This field takes one or more build status code letters. For example, to define an assertion that executes if the build status is set to any terminating status, enter the value IEF.
- **Component Build Message Expression:** the server-side component has supplied a String as the message parameter to the setBuildStatus method of the TransInfo that matches the specified regular expression.
- **Exception Type Match:**The server-side component threw and exception with a type that matches the value of this field. Possible values for the field include:
    - empty: Do not evaluate, meaning that there is never a match.
    - *: Match anything except a null.
    - anything else: Evaluated as a regular expression.
- **Max Build Time (millis):** The time to execute the server-side component took less than the number of milliseconds specified.

For more information about assertions, see the *LISA User Guide*.

Check LISA Integrator Component Content Response

The Check LISA Integrator Component Content Response assertion specifies a test case to execute if the server-side component returns content that matches a regular expression. Not every server side component can return component-level content. HTTP-based applications generally return the HTTP response.

Attributes include:

- **Component Name Spec**: The expression that selects the component whose content to search.
- **Expression**: The expression to search the component content. For both fields, the expressions are evaluated in the following way:
    - A null component value is not a hit no matter what the expression is.
    - An empty expression is never a hit.
    - A * matches any non-null value.
    - Any other expression is considered a regular expression.

The entire transaction is itself a component, so its name and content are evaluated as part of this execution.

Check LISA Integrator Reporting Missing Data

The Check LISA Integrator Reporting Missing Data assertion specifies a test case to execute if the server-side component does not return expected content. Not every server side component can return component-level content. HTTP-based applications generally return the HTTP response.

Attributes include:

- **Component Name Spec**: The expression that selects the component whose content to search.

For both fields, the expressions are evaluated in the following way:

- An empty expression is never a hit.
- A * matches any non-null value.
- Any other expression is considered a regular expression.

The entire transaction is itself a component, so its name and content are evaluated as part of this execution.

For more information about assertions, see the *LISA User Guide*.

Extending the LISA Software

This chapter explains the main concepts involved in extending the LISA software. Topics include:

- Reasons to Extend the LISA Software
- LISA Extension Concepts
- Parameters and Parameter Lists
- Test Exceptions

Reasons to Extend the LISA Software

Although the LISA software provides most of the elements needed to test enterprise software components, you may need to create your own elements to solve specific problems. For example, the LISA software does not provide built-in support for testing FTP clients. If you write an FTP client, you might create

- a custom node to test the transfer of files
- a custom node to check the contents of transferred files
- a custom filter to store portions of the file in LISA properties

You can create custom types of other LISA elements, such as data sets and companions. For more information about extending other LISA elements, see the LISA Javadoc.

LISA Extension Concepts

The following concepts are common to all LISA customization scenarios:

- lisaextensions files (replaces typemap.properties)
- wizards.xml
- Named Types
- Parameters and Parameter Lists
- The Test Exec Class
- Test Exceptions

The lisaextensions File

The **lisaextensions** file is the preferred place to define the extension points for LISA. One or more custom extensions (filters, assertions, test steps, and reports, for example) are declared in a lisaextensions file. The name of the file can be anything, but the extension needs to be **.lisaextensions** and the name must be unique relative to other lisaextensions files. For example, **example.lisaextensions** is a valid name of a lisaextensions file. This file needs to be included in a jar along with the classes that do the custom extension implementation, and the jar needs to be placed in LISA's classpath. At the time of startup, LISA looks for all files with the extension .lisaextensions and tries to read the custom extension points from them.

All custom extensions in LISA need to provide a controller, viewer (called an editor frequently), and an implementation. These are often three different classes. Often, LISA provides defaults for the controller and the viewer that you can use to avoid having to write your own. At times you may not find these defaults suitable for your needs and you can create custom versions of these.

Use of lisaextensions file

The custom element should be registered in the lisaextensions file. This subsection shows how that is accomplished. For LISA to connect a given implementation to its authoring-time controller and editor, the lisaextensions file is used.

Implementation Objects

Here is an example of the portion of the contents of a lisaextensions file for custom assertions:

```
asserts=com.mycompany.lisa.AssertFileStartsWith
```

The class name mentioned on the right side of the = sign represents the location of the classes to perform the implementation of the appropriate node logic. If more than one assertions is defined, all corresponding classes are mentioned in the same line separated by commas, as in

```
asserts=com.mycompany.lisa.AssertFileStartsWith,com.mycompany.lisa.AnotherAssert
```

Controller & Editor Objects

LISA will perform a lookup on every element declared as an implementation object to get the controller and the editor to associate with that implemention. For example, the assertion shown previously has the following additional entry in the lisaextensions file:

```
com.mycompany.lisa.AssertFileStartsWith=com.itko.lisa.editor.DefaultAssertController,com.itko.lisa.edit
```

The format of this properties file entry is

```
Implementation_class_name=controller_class_name,editor_class_name
```

You will notice that the default controller and editor are used in this example. The defaults will be good enough for most cases. However, you may come across cases where custom controllers or more frequently, custom editors may need to be provided. A lisaextensions file is the place to declare the classes corresponding to them.

The typemap.properties file

In the home directory of every LISA install there is a file named **typemap.properties**. This file includes critical information used by LISA when creating and editing test cases. The preferred mechanism for adding custom extensions to LISA is through lisaextensions file as discussed previously. The addition of custom extensions through typemap.properties is supported only for backward compatibility.

The wizards.xml file

The home directory of LISA contains a file named **wizards.xml**. This file drives many of the wizards that are rendered in LISA. Any wizard that prompts the user with frequently accessed test elements (assertions, test steps, filters, and so on) consults this file for the information to display in the tree view. Your custom LISA extensions may be added to this file to make them easily available to users, and built-in elements not used frequently may be removed. The **wizards.xml** file itself provides some useful information about how to read and modify its contents, but the concepts will be described here.

Wizard Tag

The wizard tag is used to provide a list of elements for a given type of test element and a given type of use. Following is an example:

```
<Wizard element="assert" type="http">
```

The element values (in this case "assert") and the types (in this case "http") are predefined by LISA. New element values or types will be ignored by LISA. The type is the short name for the kind of elements that should be described in the contained Entry tags. In this example, the Entry tags should be documenting assertions. The type attribute drives when LISA renders the given list. For common testing needs (like web testing, XML testing, Java testing), LISA allows the **wizards.xml** file to contain different assertions and filters that are appropriate.

Entry Tag

Here is a sample Entry tag:

```
<Entry>
   <Name>Make Diff-like Comparisons on the HTML Result</Name>
   <Help>This is the right assertion to use when you have a HTML result and you wish to perform
diff-like operations on it. It lets you define simply what portions of the text may change, what
may not change, and what portions must match current property values.</Help>
   <Type>com.itko.lisa.web.WebHTMLComparisonAssert</Type>
</Entry>
```

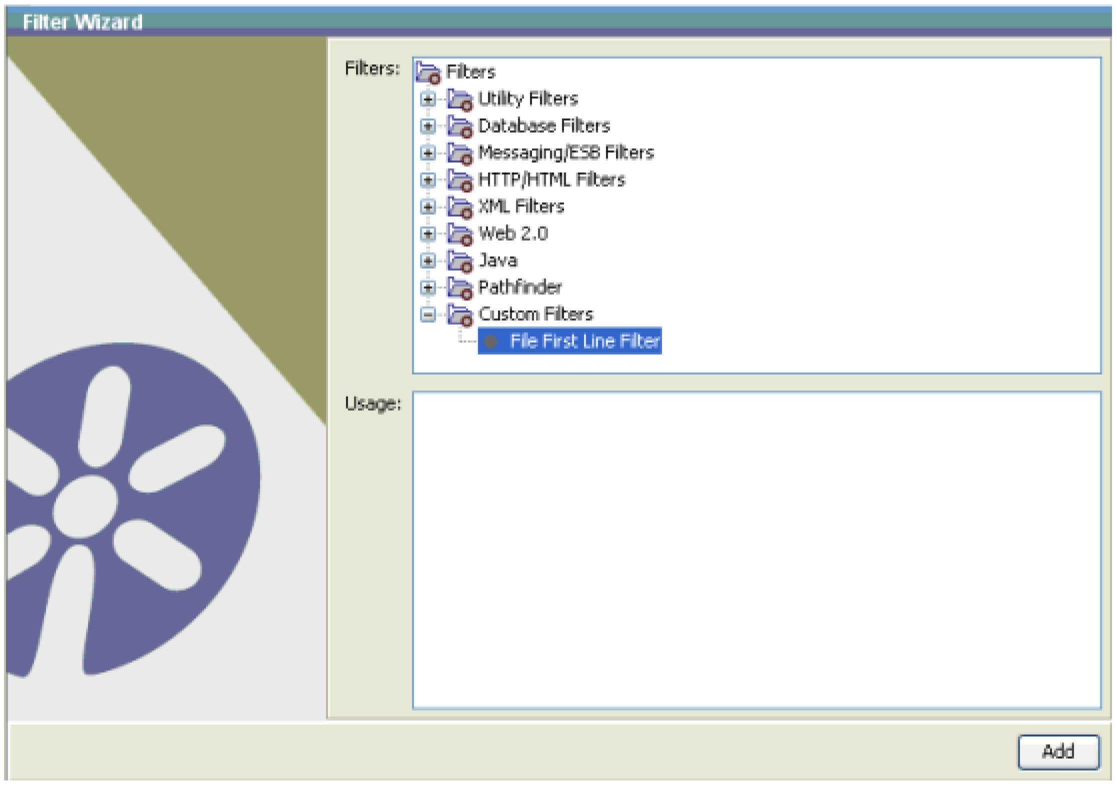
The following child tags are required to define a wizard entry:

- **Name**: This is the label used in the wizard itself, shown as a option button in the current version of LISA.
- **Help**: At the bottom of the wizard panel there is an area for more text. This text will be shown when this entry has been selected.
- **Type**: This is the actual test case element implementation class name.

The NamedType Interface

The **com.itko.lisa.test.NamedType** interface specifies that an implementing class is an object associated with a LISA object viewer. Every class you create to extend the LISA software implements this interface if it can be edited in the LISA Test Manager. It exposes one method, getTypeName, which provides the text associated with the element in the Model Editor. For example, notice the text File First Line Filter in the following image below.

NEED UPDATED SCREEN CAPTURE



The text is provided by a method in the class that defines the filter. That class must implement the **com.itko.lisa.test.NamedType** interface, and must implement the getTypeName method:

```
public String getTypeName()
 {
 return "File First Line Filter";
 }
```

Parameters and Parameter Lists

Parameters (represented by **com.itko.lisa.test.Parameter** objects) are the attributes that define and support LISA elements.

A ParameterList (represented by an object of type **com.itko.lisa.test.ParameterList**) is a collection of such parameters. In a way, ParameterList can be treated as an extension of **java.util.HashMap**.

The class that defines a LISA element defines the parameters exposed by the element in a callback method that returns a com.itko.lisa.test.ParameterList. For example, the **getParameters()** method below defines the parameters for the Parse Property Value As Argument String filter:

```
public ParameterList getParameters()
{
    ParameterList p = new ParameterList();
    p.addParameter( new Parameter( "Existing Property", PROPKEY_PARAM, propkey,
TestExec.PROPERTY_PARAM_TYPE ) );
    p.addParameter( new Parameter( "IsURL", ISURL_PARAM, new Boolean(isurl).toString(),
Boolean.class ) );
    p.addParameter( new Parameter( "Attribute", ATTRIB_PARAM, attrib, TestExec.ATTRIB_PARAM_TYPE )
);
    p.addParameter( new Parameter( "New Property", PROP_PARAM, prop, TestExec.PROPERTY_PARAM_TYPE )
);
    return p;
}
```

For each parameter exposed, the method creates a new com.itko.lisa.test.Parameter. The previous example has defined the following parameters:

- A LISA property value for Existing Property
- A Boolean value for IsURL
- A LISA attribute value for Attribute
- A LISA property for New Property

The parameters to the constructor of Parameter are:

- The string that provides the label for the parameter in the Model Editor
- A string key used to store the value of the parameter in a Java Map
- A variable that stores the value of the parameter
- The fully qualified type of the parameter

The Model Editor uses the last parameter to determine the user interface element associated with the parameter. For example, passing **Boolean.class** renders the parameter as a check box, while passing **TestExec.PROPERTY_PARAM_TYPE** renders the parameter as a drop-down menu containing the keys of all existing properties.

For more information about the **com.itko.lisa.test.Parameter** and **com.itko.lisa.test.ParameterList** classes, see the LISA Javadoc.

The TestExec Class

The com.itko.lisa.test.TestExec class provides access to the state of the test and convenience functions for performing common tasks within LISA elements. A TestExec parameter is passed to most LISA element methods. Some of the most useful methods provided by the TestExec class include:

- **log**: Fires a EVENT_LOGMSG TestEvent including the string parameter.
- **getStateObject**: Takes a string LISA property key and returns the value of that property.
- **setStateValue**: Takes a string LISA property key and an Object and sets the value of that property to the specified Object value.
- **setNextNode**: Takes the string name of a LISA node and sets the next test case to fire to that test case.
- **raiseEvent**: Allows you to fire an arbitrary LISA event.

For more information about the com.itko.lisa.test.TestExec class, see the LISA Javadoc.

Test Exceptions

The LISA software provides two main exception classes for handling errors in test case components:

- **TestRunException**: There was a problem in how the test was run. For example, trying to access a parameter that does not have an expected value would throw this type of exception.
- **TestDefException**: There was a problem in how the test was defined. For example, trying to reference a data set that was not defined would throw this type of exception.

For more information about LISA exception classes, see the LISA Javadoc.

Extending LISA Test Steps

Although the test steps provided by the LISA software provide most of the logic needed to test enterprise software, you may need to create your own test step for a specific situation. Each existing test step provides a node-specific Swing user interface to help users develop that type of test step. While you can provide this same support by creating a Java class that extends **com.itko.lisa.test.TestNode** and providing a Swing user interface, there is a much simpler way to provide a custom test step.

Many testing situations can be adequately represented by a set of name-value pairs. For example, assume that you want to test a File Transfer Protocol (FTP) client package you have written. You do not need a complex wizard to collect the information required to test. You simply need the FTP host, the full path and name of the file, and the username and password used to access the FTP host. Each of these values can be represented by a name-value pair.

LISA provides built-in support for custom test steps that fit this profile. To create a custom test step, create a Java class that implements **com.itko.lisa.test.CustJavaNodeInterface**. This class specifies which name-value pairs are associated with the test step, and the logic to run when the test step executes. At runtime, the LISA Model Editor searches the classpath for classes that implement **CustJavaNodeInterface**, and makes them available under the Custom Test Step Execution test step type.

LISA's integration kit provides two ways to enable developers to augment the functionality of LISA with new test cases:

- Custom Java Test Steps: This type of test step is faster and easier to develop, so it is often the preferred method used by developers.
- Native Test Steps: These test steps are created in precisely the way that test steps within LISA are developed by ITKO, and can appear in their own categories.

Custom Java Test Steps

This section explains how to create a custom Java test step and use it in the LISA Model Editor. This type of test step is created by extending an abstract base class provided by LISA in the Lisa.jar. It is faster and easier than the native test step in that it does not require the developer to build a UI for the Test Manager. Instead, one is auto-generated for you by invoking the calls on this class. This efficiency comes at the cost of control over how the user interacts with your test step at the time the test is being authored. Most parameters are rendered by LISA in an appropriate manner, but some parameters might require a customized editor. For those needs, consider creating a native test step.

Topics include:

- Creating a Custom Java Test Step
- Deploying a Custom Java Test Step
- Defining a Custom Java Test Step

Creating a Custom Java Test Step

To create a custom Java test step

1. Create a Java class that implements **com.itko.lisa.test.CustJavaNodeInterface**. This tells the LISA software that your class is a custom Java test step.

```
public class FTPCustJavaNode implements CustJavaNodeInterface
{

}
```

2. Implement the required **initialize** method. This is the method called when the LISA software first loads a custom test step during testing. In this example, the test step needs no initialization, so it simply logs the fact that the initialize method was called.

```
static protected Log cat = LogFactory.getLog( "com.mycompany.lisa.ext.node.FTPCustJavaNode" );

public void initialize( TestCase test, Element node ) throws TestDefException
{
    cat.debug( "called initialize" );
}
```

3. Implement the required **getParameters** method. This method specifies the name-value pairs that define the parameters to the custom test step. The simplest way to define the parameter list is to pass one long string of all parameters, with each parameter separated by an ampersand (&). Values specified here are used as default values when the node is defined in the Model Editor.

```
public ParameterList getParameters()
{
    ParameterList pl = new ParameterList(
"username=&password=&host=ftp.suse.com&path=/pub&file=INDEX" );
    return pl;
}
```

The previous example uses **ftp.suse.com/pub/INDEX** as the file to retrieve. This is a publicly available FTP host that allows anonymous login. Limit unnecessary hits on the SuSE FTP site.

4. Implement the required **executeNodeLogic** method. This is where you define the logic that runs when the test step executes. Typically, this is used to instantiate and validate components of the system under test.
The TestExec parameter provides access to the test environment, such as logs and events. The Map parameter provides access to the current value of the test step parameters. The Object return type allows you to pass data back to the running test, so that you can run assertions and filters on it.

```
public Object executeNodeLogic( TestExec ts, Map params ) throws TestRunException
{
    ts.log( "We got called with: " + params.toString() );
    String host = (String)params.get("host");
    String username = (String)params.get("username");
    String password = (String)params.get("password");
    String path = (String)params.get("path");
    String file = (String)params.get("file");
    String storedFile = runFTP(ts,host,username,password,path,file);
    FileDataObject fdo = new FileDataObject(storedFile);
    return fdo;
}
```

It is common to use a custom data object for the return value. This strategy has two benefits:

- It encourages proper resource handling. Common resources, such as files and JDBC connections, should not be passed as results, as the node does not get a chance to clean up by calling a close method. If you construct your own data object, you can store and pass just the information you need from the resource, then close the resource before returning the data object.
- It allows you to perform conditional filtering and result-checking based on the type of the data object. For more information on using data objects to perform conditional filtering, see Creating a New LISA Filter.

The previous code uses the FileDataObject custom data object to store just the path to the stored file, rather than passing an open File or FileInputStream. Any filters can perform conditional processing by checking that the type of the result object is FileDataObject.

Deploying a Custom Java Test Step

Before you can use a custom Java test step in a LISA test case, you must make it available to the LISA Model Editor.

To deploy a custom Java test step

1. Create a lisaextensions file or use an existing one if one exists. In lisaextensions file, mention this class against simpleNodes element.

```
simpleNodes=com.mycompany.lisa.FTPCustJavaNode
```

2. Copy the JAR file that contains your custom Java test step and the lisaextensions file to the LISA hotDeploy directory at **LISA_HOME/hotDeploy**. If your custom Java test step depends on any third-party libraries, copy those to the LISA hotDeploy directory.

For this example, the FTPCustJavaNode described previously has been packaged for you at **LISA_HOME/doc/DevGuide/lisaint-examples.jar**. This custom Java test step depends on the FTP client packaged at **LISA_HOME/doc/DevGuide/lib/ftp.jar**.

Copy both of these files to the LISA hotDeploy directory.

⚠ The FTP client used in the sample code is provided by www.amoebacode.com/.

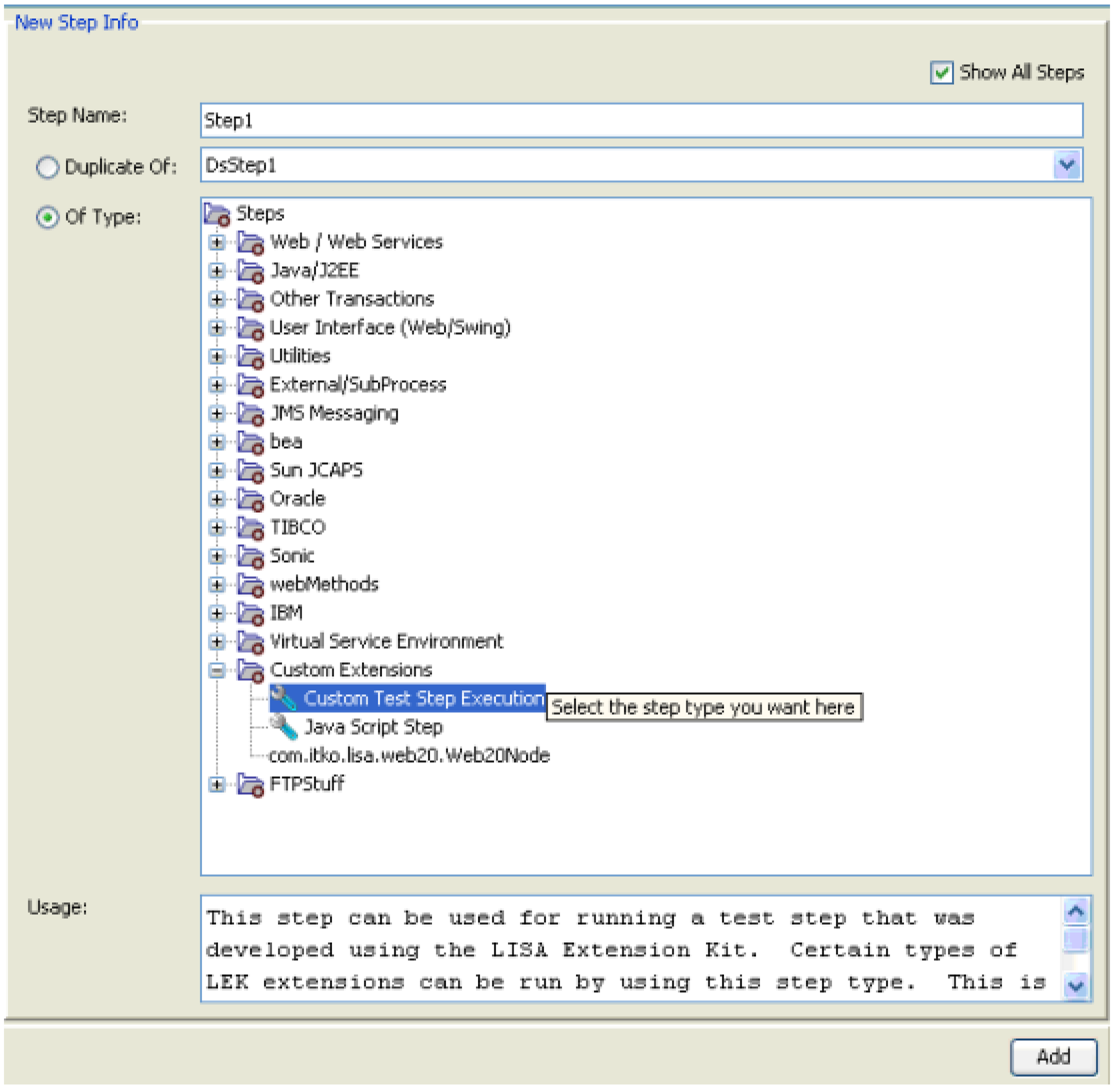
This will put the given class name in a convenient drop-down list for users when authoring a test case. It is optional because LISA can be given the class name at authoring time by typing or using the Class Path Navigator.

If you are already running LISA, you will need to exit and restart the program for this new setting to take effect.

Defining a Custom Java Test Step

To define a custom Java test step in the LISA Model Editor

1. Change the **Type** of the node to **Custom Test Step Execution**:


New Step Info

[✓] Show All Steps

Step Name: Step1

Duplicate Of: DsStep1

[✓ Of Type:]

```
Steps
+ Web / Web Services
+ Java/J2EE
+ Other Transactions
+ User Interface (Web/Swing)
+ Utilities
+ External/SubProcess
+ JMS Messaging
+ bea
+ Sun JCAPS
+ Oracle
+ TIBCO
+ Sonic
+ webMethods
+ IBM
+ Virtual Service Environment
- Custom Extensions
    Custom Test Step Execution   Select the step type you want here
    Java Script Step
    com.itko.lisa.web20.Web20Node
+ FTPStuff
```

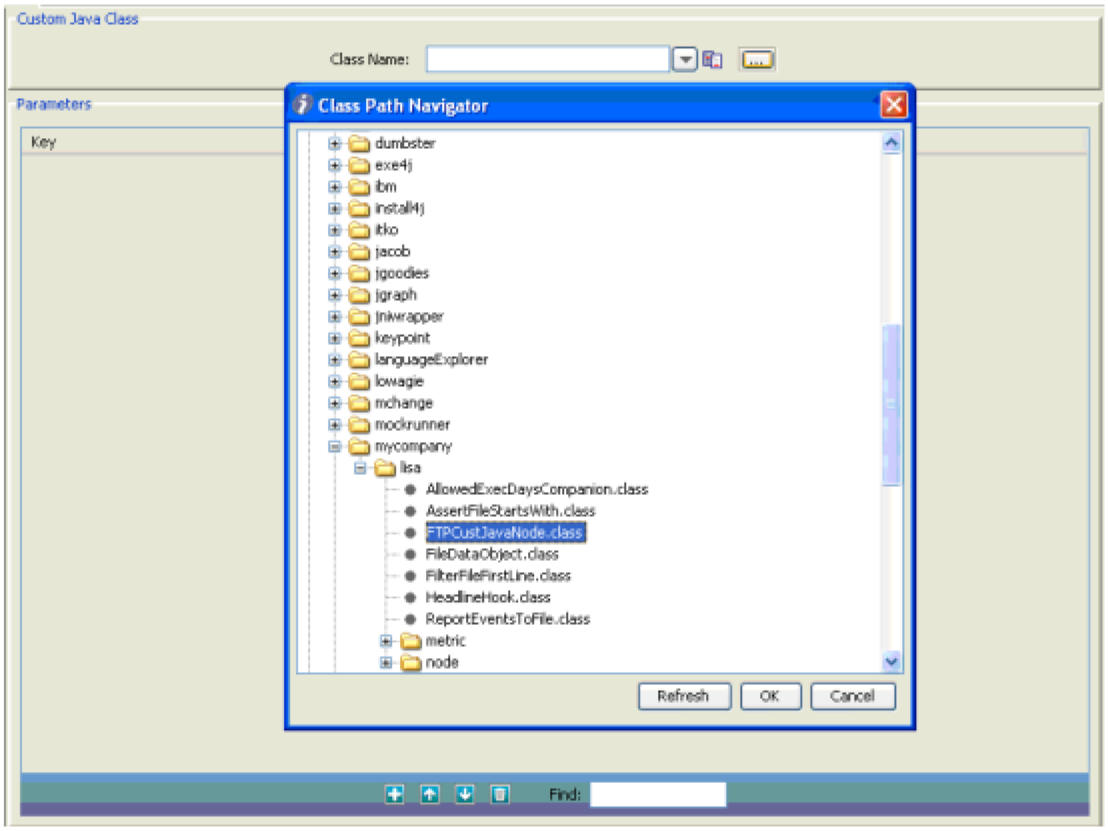
Usage:

```
This step can be used for running a test step that was
developed using the LISA Extension Kit.  Certain types of
LEK extensions can be run by using this step type.  This is
```

Add

2. Specify the custom test step to test. Enter or select the fully qualified name of the Java class in the Class Name field. To easily specify a custom Java test step class, click the ellipsis ( ) next to the Class Name field to launch the Class Path Navigator, which only looks for custom test steps. Select the test step class to test and click OK.

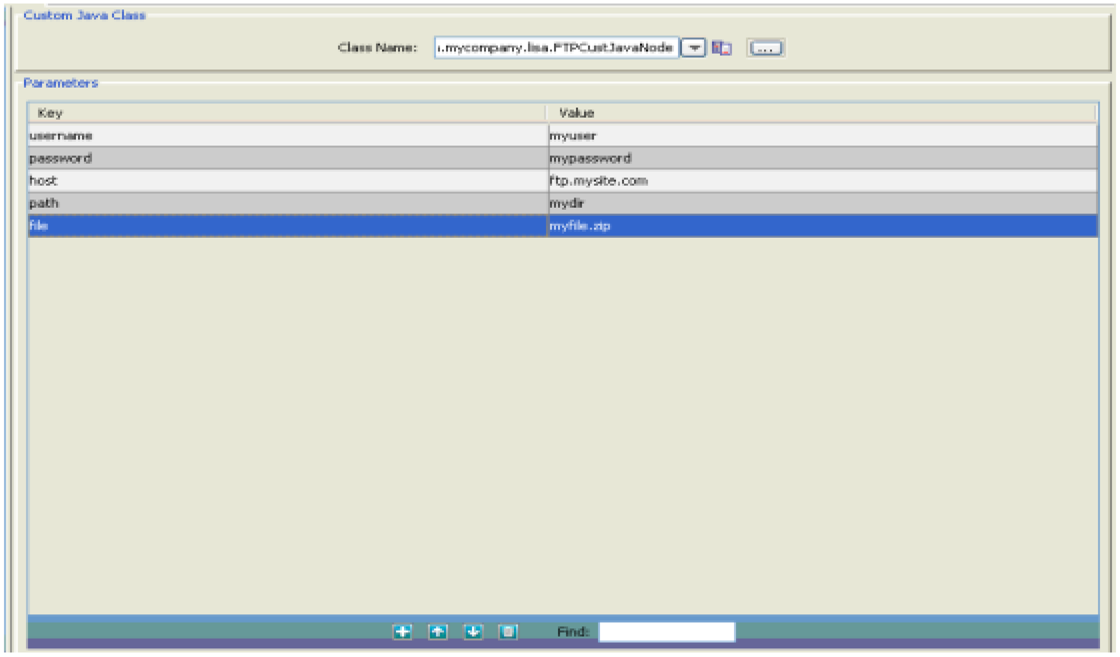
NEED UPDATED SCREEN CAPTURE



```
dumbster
exe4j
ibm
install4j
itko
jacob
jgoodies
jgraph
jniwrapper
keypoint
languageExplorer
lowagie
mchange
mockrunner
mycompany
  lisa
    AllowedExecDaysCompanion.class
    AssertFileStartsWith.class
    FTPCustJavaNode.class
    FileDataObject.class
    FilterFileFirstLine.class
    HeadlineHook.class
    ReportEventsToFile.class
    metric
    node
```

3. Set parameters to the custom test step. For each parameter in the Parameters list, supply a value.

NEED UPDATED SCREEN CAPTURE



Class Name: ı.mycompany.lisa.FTPCustJavaNode

Key	Value
username	myuser
password	mypassword
host	ftp.mysite.com
path	mydir
file	myfile.zip

This example uses ftp.mysite.com/mydir/myfile.zip as the file to retrieve, using credentials myuser/mypassword

Native Test Steps

This section explains how to create a native test step and use it in the LISA Model Editor. The native test step is the way to provide 100% customized test step authoring and execution for your test steps. Instead of the Custom Java Step defined previously, you are responsible for providing the editing environment and the complete execution engine for your test step. This is precisely how native test steps provided as built-in within LISA are developed.

Topics include:

- Creating a Native Test Step
- Deploying a Native Test Step
- Defining a Native Test Step

Creating a Native Test Step

To create a native test step

1. Create a Java class that extends **com.itko.lisa.test.TestNode**. The runtime logic of the test step will be implemented by this class. See the Javadoc for TestNode and the sample code provided for FTPTestNode in the samples that accompany this document. The following calls are important for proper construction of a test step:

```
public void initialize( TestCase test, Element node )
    throws TestDefException
```

LISA invokes this call for you to read the parameters you require for operation from the test case XML DOM. The test case as it is being constructed is passed along with the DOM Element of the test step that represents this node in the test case XML.

```
public void execute( TestExec ts )
    throws TestRunException
```

This is called when your test step logic needs to be invoked. LISA's workflow and state engines manage most of the control flow and data requirements for you. You can access the TestExec given as a parameter to perform a variety of tasks. See the Javadoc on TestExec and the description of this class in Integrating Components.

2. Create a Java class that extends **com.itko.lisa.editor.TestNodeInfo**. To create and edit your node, you must provide a controller and viewer in the MVC pattern. The TestNodeInfo is the base class for all test steps that are developed for LISA. The LISA authoring framework executes this class to interact with your test step's data and logic. See the Javadoc for TestNodeInfo and especially look at the example class provided called **FTPNodeController**.

3. Create a Java class that extends **com.itko.lisa.editor.CustomEditor**. This class provides LISA with the actual UI for viewing and editing your node's data. This class with be constructed and called by the authoring framework to display your parameters, check their validity, and save your changes back into the TestNodeInfo extension described previously. This CustomEditor extends JPanel in the Java Swing API. See the Javadoc for CustomEditor and view the sample code for help writing your own editors.

Deploying a Native Test Step

Native test steps must be explicitly declared to LISA at startup of the Test Manager so that the authoring framework can associate the three classes defined previously. This declaration is done in the lisaextensions file against the nodes element. The classes themselves do not refer to one another. They are connected in the lisaextensions file.

The following example defines a new category for the custom test step. It will be used in the tree of the UI. You must precede any spaces in a node category with the backslash character.

```
nodes=com.mycompany.lisa.node.FTPTestNode
 nodeCategories=FTP\ Stuff
 FTP\ Stuff=com.mycompany.lisa.node.FTPTestNode
 com.mycompany.lisa.node.FTPTestNode=com.mycompany.lisa.node.FTPTestNodeController,com.mycompany.lisa.n
```

For more information about the lisaextensions file, see Extending the LISA Software.

Defining a Native Test Step

Your native test step can be accessed in exactly the same way that existing, built-in nodes are made available in LISA. See the LISA User Guide for information on how to access a test step.

Extending LISA Assertions

This chapter explains how to extend the LISA software with a new LISA assertion. Topics include:

- Creating a New LISA Assertion
- Deploying a New LISA Assertion
- Defining and Testing a New LISA Assertion

Creating a New LISA Assertion

Although the assertions provided by the LISA software provide most of the logic needed to test enterprise software, you may need to create your own assertion for a specific situation. The LISA software provides built-in support for custom assertions. To create a custom assertion, create a Java class that extends **com.itko.lisa.test.CheckResult**. This class handles most of the behind-the-scenes logic required to execute the assertion, and provides a nice user interface in the Model Editor.

To create a custom assertion

1. Create a Java class that extends **com.itko.lisa.test.CheckResult**. This tells the LISA software that your class is a custom assertion and provides you with the needed assertion functionality.

```
public class AssertionFileStartsWith extends CheckResult
{

}
```

2. Implement the required **getTypeName** method. This is the method that provides the name used to identify the custom assertion in the Model Editor.

```
public String getTypeName()
{
    return "Results File Starts With Given String";
}
```

3. Handle custom parameters to the assertion. If the assertion requires more information than that provided in the node execution result, you must provide a parameter so the user can specify that information on the Assertions tab of the Model Editor.

Implement the abstract getCustomParameters method, in which you create a new ParameterList and add a Parameter for each parameter to the assertion. The constructor for the Parameter takes:

- A string that provides the label for the parameter in the user interface.
- A string that provides the key to store the value of the parameter in a Map.
- A string representing the value of the parameter.
- The type of the parameter.

In this example, the custom assertion takes one parameter, the string to find in the file. The rest of the code provides support for the parameter.

```
public static final String PARAM = "param";

private String param = "";

...

public String getParam()
{
    return param;
}

public ParameterList getCustomParameters()
{
    ParameterList pl = new ParameterList();
    pl.addParameter( new Parameter( "Starts With String", PARAM, param, String.class ) );
    return pl;
}
```

4. Initialize the custom assertion object with the value of the DOM Element. When LISA tries to execute an assertion, it first creates an instance of the custom class and then calls the **initialize** method, passing the XML element that defined the assertion. You must store the values of the parameter child elements in the new instance. For example, the test case XML may include an assertion that looks like this:

```
<CheckResult name="CheckThatFile"
 log="Check if the node FTP'd a file that starts with 'All Visitors'"
 type="com.mycompany.lisa. AssertFileStartsWith" >
 <then>finalNode</then>
 <param>All Visitors</param>
 </Assertion>
```

In the **initialize** method, you need to get the text "All Visitors" into the param member variable. The following code grabs the text from the child element named param and checks to make sure it is not null.

```
public void initialize( Element rNode ) throws TestDefException
{
    param = XMLUtils.getChildText( XMLUtils.findChildElement( rNode, PARAM ) );
    if( param == null )
    throw new TestDefException( rNode.getAttribute("name"),
    "File Starts With results must have a Starts With String parameter specified.", null );
}
```

You are free to use whatever means you wish to read from the DOM Element, but you can use a convenience class named **XMLUtils**, as seen in the previous example.

5. Implement the checking logic with the **evaluate** method. The TestExec parameter provides access to the test environment, such as logs and events. The Object parameter provides access to result returned from executing the node. The Boolean return type returns true if the assertion is true, false otherwise.

The following code checks that the first line of the file passed from the node contains the string stored in the param parameter. It returns true only if the file starts with that string.

```
public boolean evaluate( TestExec ts, Object oresult )
{
    if( oresult == null )
    return false;
    FileDataObject fdo = (FileDataObject)oresult;
    String firstline = fdo.getFileFirstLine();
    return firstline.startsWith(param);
}
```

As an author of an assertion, your job is simply to declare whether the assertion as defined is true or false, and not to worry about the steps to execute after the state is returned. Further, the LISA workflow engine will evaluate if that is considered a fired assertion or not.

Deploying a New LISA Assertion

Before you can use a custom assertion in a LISA test case, you must make it available to the Model Editor.

To deploy a custom assertion

1. Tell LISA to look for a new custom assertion in **lisaextensions** file, as:

```
asserts=com.mycompany.lisa.AssertFileStartsWith
 com.mycompany.lisa.AssertFileStartsWith=com.itko.lisa.editor.DefaultAssertController,com.itko.lisa.edi
```

You may also add this assertion to the wizards in the **wizards.xml** file.

2. Copy the JAR file that contains your custom assertion and **lisaextensions** file to the LISA hotDeploy directory at **LISA_HOME/hotDeploy**. If your custom assertion depends on any third-party libraries, copy those to the LISA hotDeploy directory. In this example, the AssertionFileStartsWith described previously has already been packaged for you at **LISA_HOME/doc/DevGuide/lisaint-examples.jar**. This custom assertion does not depend on any third-party libraries.

3. If you are already running LISA, you need to exit and restart the program for this new setting to take effect.

Defining and Testing a New LISA Assertion

To define a custom assertion

1. Change the Type of the assertion, selecting the text you specified in the **getTypeNam**e method.
2. Set parameters to the custom assertion. For each parameter in the **Assertion Param Attributes** section, supply a value.
3. Use a custom assertion like you would any built-in assertion.

Extending LISA Filters

This chapter explains how to extend the LISA software with a new LISA filter. Topics include:

- Creating a New LISA Filter
- Deploying a New LISA Filter
- Defining and Testing a New LISA Filter

Creating a New LISA Filter

Although the filters provided by the LISA software provide most of the logic needed to test enterprise software, you may need to create your own filter for a specific situation. The LISA software provides built-in support for custom filters.

To create a custom filter

1. Create a Java class that implements **com.itko.lisa.test.FilterInterface**. This tells the LISA software that your class is a custom filter.

```
public class FilterFileFirstLine implements FilterInterface
{

}
```

2. Implement the required **getTypeName** method. This is the method that provides the name used to identify the custom filter in the Model Editor

```
public String getTypeName()
{
    return "File First Line Filter";
}
```

3. Define parameters to the filter. For each item in the Filter Attributes section of the Filters tab in the Model Editor, you must add a Parameter to

the filter's ParameterList. In this example, the custom filter takes two parameters:

- A check box to identify if the text in the first line of the file represents an FTP host.
- A text box to identify the parameter in which the line is stored.

The following code creates the two parameters:

```
public ParameterList getParameters()
{
    ParameterList p = new ParameterList();
    p.addParameter( new Parameter( "Is FTP", ISFTP_PARAM, new Boolean(isftp).toString(),
Boolean.class ) );
    p.addParameter( new Parameter( "New Property", PROP_PARAM, prop,
TestExec.PROPERTY_PARAM_TYPE));
    return p;
}
```

4. Initialize the custom filter object with the value of the DOM Element. When LISA tries to execute a filter, it first creates an instance of the custom class and then calls the **initialize** method, passing the XML element that defined the filter. You must store the values of the parameter child elements in the new instance. For example, the test case may include a filter that looks like this:

```
<Filter type="com.mycompany.lisa.ext.filter.FilterFileFirstLine"
 isftp="true" prop="THE_LINE" />
```

In the **initialize** method, you need to set the **isftp** variable to true and the **prop** variable to THE_LINE.

```
static private String ISFTP_PARAM = "isftp";
static private String PROP_PARAM = "prop";
private String prop;
private boolean isftp;
//...

public void initialize( Element e ) throws TestDefException
{
    try {
        String s = XMLUtils.findChildGetItsText( e, ISFTP_PARAM ).toLowerCase();
        if( s.charAt(0) == 'y' || s.charAt(0) == 't' )
            isftp = true;
        else
            isftp = false;
    }
    catch( Exception ex ) {
        isftp = false;
    }
    prop = XMLUtils.findChildGetItsText( e, PROP_PARAM );
    if( prop == null || prop.length() == 0 )
    prop = "FILE_FIRST_LINE";
}
```

This code uses a utility class in lisa.jar named **com.itko.util.XMLUtils** to find child tags of the given parent tag and read the child text of the tag. This approach is useful because LISA automatically writes the XML representation of filters by making each of the Parameter objects in the getParameters a child tag of the Filter tag. Each parameter key becomes the tag name and the child text of the tag is the value.

5. Because filters execute before and after the test step, you get two chances to implement filter logic. Implement filter logic before node execution with the **preFilter** method. The TestExec parameter provides access to the test environment, such as logs and events. The Boolean return type returns false if the filter did not set a new node to execute, true if it did.

In this example, we are not interested in filtering before the node executes, so the pre-filter method does nothing.

```
public boolean preFilter( TestExec ts ) throws TestRunException
{
    // don't have anything to do...
    return false;
}
```

Implement filter logic after node execution with the **postFilter** method. The TestExec parameter provides access to the test environment, such as logs and events. The Boolean return type returns false if the test should continue as normal, true otherwise.

In this example, we store the first line of the file returned from the node in the new property, prepending ftp:// if the **isFTP** box was checked.

A LISA user can use a filter at either the test step level or the test case level. Add logic to the filter to check if the result is the proper state to run the filter. For example, if your filter assumes that the LASTRESPONSE holds a FileDataObject, then check for that before executing the filter logic.

```
public boolean postFilter( TestExec ts ) throws TestRunException
{
    try {
        Object oresponse = ts.getStateObject( "LASTRESPONSE" );
        if (!(oresponse instanceof FileDataObject))
            return false;
        FileDataObject fdo = (FileDataObject)oresponse;
        String firstline = fdo.getFileFirstLine();
        if((firstline ==null) || (firstline.equals(""))) {
            ts.setStateValue( prop, "" );
            return false;
        }
        if( isftp )
            firstline="ftp://" + firstline;
        ts.setStateValue( prop, firstline );
        return false;
    }
    catch( Exception e )
    {
        throw new TestRunException( "Error executing FilterFileFirstLine", e );
    }
}
```

6. Implement the **getNodeConnections** method. This method is used at test case authoring time to inform LISA of what possible test case nodes can be referenced by this filter. For example, a filter that would set the next node to "fail" on a given condition would want to make a NodeConnection object to encapsulate that reference. See the **com.itko.lisa.test.NodeConnection** in the Javadoc for more on this class. This is also how test elements are informed when the name of a test step is changed.

```
public Collection getFilterNodeConnections()
{
    return null;
}
```

Deploying a New LISA Filter

Before you can use a custom filter in a test case, you must make it available to the Model Editor.

To deploy a custom filter

1. Tell LISA to look for a new custom filter in a **lisaextensions** file, as:

```
filters=com.mycompany.lisa.FilterFileFirstLine
 com.mycompany.lisa.FilterFileFirstLine=com.itko.lisa.editor.FilterController,com.itko.lisa.editor.Defa
```

The filter may be added to the wizards in the **wizards.xml** file.

2. Copy the JAR file that contains your custom filter and **lisaextensions** file to the LISA hotDeploy directory at **LISA_HOME/hotDeploy**. If your custom filter depends on any third-party libraries, copy those to the LISA hotDeploy directory. In this example, the FilterFileFirstLine described previously has already been packaged for you at **LISA_HOME/doc/DevGuide/lisaint-examples.jar**. This custom filter does not depend on any third-party libraries.

3. If you are already running LISA, you need to exit and restart the program for this new setting to take effect.

Defining and Testing a New LISA Filter

To define a custom filter in the Model Editor

1. Change the **Type** of the filter, selecting the text you specified in the **getTypeName** method.
2. Set parameters to the custom filter. For each parameter in the **Filter Attributes** section, supply a value.
3. Test a custom filter just like you would any built-in filter.

Custom LISA Reports

This chapter explains how to extend the LISA software with new LISA reports. Topics include:

- Creating a New LISA Report Generator
- Deploying a New LISA Report Generator
- Using a New LISA Report Generator

Creating a New LISA Report Generator

Although the report generators provided by the LISA software provide most of the output needed, you may need to create your own report for a specific situation. The LISA software provides built-in support for custom report generators.

To create a custom report

1. Create a Java class that extends **com.itko.lisa.coordinator.ReportGenerator**. This tells the LISA software that your class is a custom report.

```
public class ReportEventsToFile extends ReportGenerator
 {
 }
```

2. Implement the required getTypeName method. This is the method that provides the name used to identify the custom report in the Staging Document Editor.

```
public String getTypeName()
 {
 return "Report Events To a File";
 }
```

3. Define parameters to the report. For each item in the **Report Attributes** section of the **Reports** tab in the Staging Document Editor, you must add a Parameter to the report's ParameterList.

4. Initialize the custom report generator object with the ParameterList given. When LISA tries to execute a report, it first creates an instance of the custom class and then calls the initialize method, passing the ParameterList that was read from the XML of the staging document. LISA automatically reads and writes the XML representation of report attributes by making each of the Parameter objects in the getParameters a child tag of the report XML tag. Each parameter key becomes the tag name and the child text of the tag is the value.

5. While the test is running, LISA will invoke the pushEvent method for every event you have not filtered.

6. Implement the finished method. When LISA has finished the test, it will invoke this method on your report generator. This is your opportunity to complete your processing, like saving the document you have been writing.

Deploying a New Report Generator

Before you can use a custom report in a LISA staging document, you must make it available.

To deploy a custom report

1. Tell LISA to look for a new custom report generator in a lisaextensions file, as:

```
reportGenerators=com.mycompany.lisa.ReportEventsToFile
```

The report may also be added through lisa.properties file using **lisa.editor.reportGenerators key**.

2. Copy the JAR file that contains your custom report and the lisaextensions file to the LISA hotDeploy directory at %LISA_HOME%\hotDeploy. If your custom report depends on any third-party libraries, copy those to the LISA hotDeploy directory. If you are already running LISA, exit and restart the program for this new setting to take affect.

Using a New Report Generator

To use a custom report in the LISA Staging Document Editor, access it the same way as any other built-in report.

Custom Report Metrics

This chapter explains how to extend the LISA software with a new reporting metrics. Topics include:

- Creating a New LISA Report Metric
- Deploying a New LISA Report Metric

Creating a New LISA Report Metric

When a test case is staged, a subsystem within LISA samples metric values and reports them in the ways defined in the staging document. The staging document includes the metrics that should be collected, or users can add them while the test runs.

Two classes must be created for metric collection:

- The **Metric Integrator** provides LISA with a way to view and edit the metrics that you want to collect.
- The **Metric Collector** is the engine that samples values while a test is running.

To create a custom report

1. Create a Java class that implements **com.itko.lisa.stats.MetricIntegration**. This class gives LISA the metrics that you want to access during the staging of a test.

```
public class RandomizerMetricIntegration implements MetricIntegration
{

}
```

2. Implement the required **getTypeName** method. This is the method that provides the name used to identify the custom report metric type in the Staging Document Editor.

```
public String getTypeName()
{
    return "Randomizer Metric";
}
```

3. Implement the **public MetricCollector[] addNewCollectors(Component parent)** method so that at design time your code can define the requested metrics to collect.

4. Create a Java class that extends **com.itko.lisa.stats.MetricCollector**. LISA calls on instances of this class to collect the metrics requested. The class must also implement **Serializable**.

```
public class RandomMetricCollector extends MetricCollector implements java.io.Serializable
{

}
```

5. Complete the implementation of your **MetricIntegration** and **MetricCollector** objects. See the Javadoc of those classes and the sample code for **RandomizerMetricIntegration** and **RandomMetricCollector** for more information about the individual methods available to extend.

Deploying a New Report Metric

Before you can use a custom report metric in a LISA staging document, you must make it available.

To deploy a custom report metric

1. Tell LISA to look for a new custom report metric in a **lisaextensions** file, as:

```
metrics=com.mycompany.lisa.metric.RandomizerMetricIntegration
```

The report metric may also be added through **lisa.properties** file using **stats.metrics.types** key.

2. Copy the JAR file that contains your metric and the **lisaextensions** file to the LISA hotDeploy directory at **LISA_HOME/hotDeploy**. If your custom metric depends on any third-party libraries, copy those to the LISA hotDeploy directory.

If you are already running LISA, you will need to exit and restart the program for this new setting to take effect.

Custom Companions

This chapter explains how to extend the LISA software with a new companion. Topics include:

- Creating a New LISA Companion
- Deploying a New LISA Companion

There are two ways to create custom companions for LISA. Native companions are created in much the same way that Native test steps are created. And somewhat like the Custom Java Node, there is a simpler way in which LISA shields you from most of the editor and serialization overhead. That is the approach we will document here.

Creating a New LISA Companion

1. Create a Java class that extends **com.itko.lisa.test.SimpleCompanion**. This class provides LISA all the information that is needed to create, edit, and execute your companion's logic.

```
public class AllowedExecDaysCompanion extends SimpleCompanion implements Serializable
{

}
```

Make sure you implement Serializable as this is important when your companion will be used in remotely staged tests.

2. Implement the required **getTypeName** method. This is the method that provides the name used to identify the companion in the Model Editor.

```
public String getTypeName()
{
    return "Execute Only on Certain Days";
}
```

3. Implement the **getParameters** method. This is the method that provides LISA with the parameters you need for your companion to be executed. You must call the superclass implementation of this method as well. The parameters shown here are made editable by the Model

Editor, and are given to you at the time of execution. You do not have to implement this method if your companion requires no parameters.

```
public ParameterList getParameters()
{
    ParameterList pl = super.getParameters();
    pl.addParameter( new Parameter( "Allowed Days (1=Sunday): ", DAYS,
        "2,3,4,5,6", String.class ) );
    return pl;
}
```

More information on Parameters and ParameterLists can be found in Extending the LISA Software.

4. Implement the **testStarting** method. This is the method called by LISA with the parameters you requested. If an error occurs or you otherwise wish to prevent the test from executing normally, throw a **TestRunException**.

```
protected void testStarting( ParameterList pl, TestExec testExec )
    throws TestRunException
```

5. Implement the **testEnded** method. This is the method called by LISA with the parameters you requested. You have an opportunity to perform any post-execution logic for the test.

```
protected void testEnded( ParameterList pl, TestExec testExec )
    throws TestRunException
```

Deploying a New LISA Companion

Companions must be explicitly declared to LISA at startup of LISA so that the authoring framework can make the companion available for use in test cases.

Like the Native Test Node, three classes are required for companions, but LISA provides default implementations for the two classes not documented here. The default controller is **com.itko.lisa.editor.CompanionController** and the editor is named **com.itko.lisa.editor.SimpleCompanionEditor**. You will notice that LISA's built-in companion called the **Set First/Last Node Companion** is actually defined with these classes, so use that companion's registration as a sample for you. They are connected in the **lisaextensions** file.

```
companions=com.mycompany.lisa.AllowedExecDaysCompanion
com.mycompany.lisa.AllowedExecDaysCompanion=com.itko.lisa.editor.CompanionController,com.itko.lisa.edit
```

See Extending the LISA Software on the **lisaextensions** file for details.

As with all custom LISA test elements, you will need to make the classes you have developed and the **lisaextensions** file available to LISA. The most common way to do this is to put a jar file in LISA's hot deploy directory.

✓ A custom companion can implement the StepNameChangeListener interface and thereby be notified if any step's name was changed. An SDK developer will do this if your custom companion renders a drop-down list of the steps in the test. An example is the Final Step to Execute companion, which lists the step names so a test author can choose the teardown step. This code change was added to enable notifying the Final Step to Execute companion when a step name changes.

Using LISA Hooks

This chapter explains how to extend the LISA software with a new hook. Topics include:

- Creating a New LISA Hook
- Deploying a New LISA Hook

A *hook* is a mechanism that allows for the automatic inclusion of test setup and/or teardown logic for all the tests running in LISA. An alternate definition of a hook is a system-wide companion.

Hooks are used to configure test environments, prevent tests from executing that are not properly configured or do not follow defined best practices, or simply to provide common operations.

Anything that a hook can perform can be instead modeled as a companion in LISA. However, these are the differences between hooks and companions:

- Hooks are global in scope. Users do not specifically include a hook in their test case as is the required practice for companions. If you need every test to include the logic and do not want users to accidentally not include it, a hook is a better mechanism.
- Companions can have custom parameters and are rendered in the Model Editor, while hooks are practically invisible to the user and therefore can request no special parameters. Hooks instead get their parameters from properties in the configuration or from the system.
- Hooks are deployed at the LISA install level, not at the test case level. If a test is run on two computers, and one computer has a hook registered and the other does not, then the hook will run only when the test is staged on the computer where it is explicitly deployed. Companions defined would execute regardless of any install-level configuration. See the following for information about how hooks are deployed.

Creating a New LISA Hook

1. Create a Java class that extends **com.itko.lisa.test.Hook**. This class gives LISA all the information that is needed to execute your hook's logic.

```
public class HeadlineHook extends Hook
{

}
```

2. Implement the **startupHook** method. This is the method called by LISA when the test is starting. If an error occurs or you otherwise wish to prevent the test from executing normally, throw a **TestRunException**.

```
public void startupHook( TestExec testExec ) throws TestRunException
```

3. Implement the **endHook** method. You have an opportunity to perform any post-execution logic for the test.

```
public void endHook( TestExec testExec )
```

Deploying a New LISA Hook

Hooks are deployed by being registered by class name in the **lisa.properties** file. The system property key that is used for hooks is **lisa.hooks**. An example follows:

```
# to register hooks with LISA, these are comma-separated
lisa.hooks=com.itko.lisa.files.SampleHook,com.mycompany.lisa.HeadlineHook
```

The preceding **lisa.properties** entry deploys two hooks to be run on every test.

Custom LISA Data Sets

This chapter explains how to extend the LISA software with a new data set. Topics include:

- Data Set Characteristics
- Creating a New LISA Data Set
- Deploying a New LISA Data Set

Data sets are created in much the same way that test steps are created. There is a simpler way in which LISA shields you from most of the editor and serialization overhead by using default class implementations for the controller and editor. That is the approach we will document here.

Data Set Characteristics

Data sets are different than every other extension mechanism in LISA in that they are inherently remote server objects. Consider a load test that has thousands of virtual users all trying to access the same spreadsheet file. LISA must create a single object that is serving the spreadsheet for all those virtual users to read.

The infrastructure for remoting your custom data set is provided automatically by LISA, so there are likely to be no specific issues related to this unique characteristic other than the following: Because they are shared, they have no access to an individual test case or test execution state. This means that you will not have access to a TestCase or TestExec object. Your class will be run in the address space of the coordinator that is staging the test, not necessarily the simulator that is communicating with the system under test.

Creating a New LISA Data Set

1. Create a Java class that extends **com.itko.lisa.test.DataSetImpl**. This class gives LISA all the information that is needed to execute your data set logic.

```
public class SomeDataSet extends DataSetImpl
 {
 }
```

Realize that your data set object will be a Remote RMI object, and will therefore be able to throw a RemoteException from its constructor and some methods.
2. Implement the required **getTypeName** method. This is the method that provides the name used to identify the companion in the LISA Test Case Editor.

```
public String getTypeName()
 {
 return "Nifty Data Set";
 }
```

3. Implement the getParameters method. This is the method that provides LISA with the parameters you need for your companion to be executed. You must call the super class implementation of this method as well. The parameters shown here are made editable by the LISA Test Case Editor, and are given to you at the time of execution. You do not have to implement this method if your companion requires no parameters.

```
public ParameterList getParameters() throws RemoteException
{
ParameterList pl = super.getParameters();
// …
return pl;
}
```

More information about Parameters and ParameterLists may be found in Extending the LISA Software.
4. Implement the two required initialize methods. These methods are provided so that the data set can be initialized from either XML or the ParameterList system within LISA.

```
public void initialize(Element dataset)
 throws TestDefException
 public void initialize(ParameterList pl, TestExec ts)
 throws TestDefException
```

5. Implement the getRecord method. This is the method called by LISA when another row is needed from the data set's data source. If an error occurs or you otherwise wish to prevent the test from executing normally, throw a TestRunException.

```
synchronized public Map getRecord()
 throws TestRunException, RemoteException
```

If you are out of rows in the data source, you must specifically code for the two possible conditions that the user may want:

1. Restart reading the data source from the top again automatically, or
2. Return a null from this method to note that fact so that the test workflow will reflect the condition.

Psuedo-code for this might look like the following:

```
Read next row
 If no-next-row, Then
 If there is no "at end" parameter specified, Then
 Re-open the data source
 Read next row
 Else
 Return null
 Return row values
```

Deploying a New LISA Data Set

Data sets must be explicitly declared to LISA at startup of the LISA Workstation so that the authoring framework can make the data set available for use in test cases. Like the Native Test Node, three classes are required for data sets, but LISA provides default implementations for the two classes not documented here. The default controller is **com.itko.lisa.editor.DataSetController** and the editor is named **com.itko.lisa.editor.DefaultDataSetEditor**. You will notice that some of LISA's built-in data sets use these classes.

They are connected in the lisaextensions file, as

```
datasets=com.itko.examples.dataset.CSVDataSet
 com.itko.examples.dataset.CSVDataSet=com.itko.lisa.editor.DataSetController,com.itko.lisa.editor.Defau
```

See the information about the lisaextensions file in Extending the LISA Software for more details on how to register your data set.

As with all custom LISA test elements, you will need to make the classes you have developed and the lisaextensions file available to LISA. The most common way to do this is to bundle them together in a jar file and put it in LISA's hot deploy directory.

LISA Java .NET Bridge

On Windows, LISA embeds a library that enables in-process bi-directional communication between the Java VM and the CLR (.NET). This library is named jdbridge (as in **j**ava **d**otnet **bridge**) and is made of three components: **jdbridge.jar** (the Java-side stubs), **djbridge.dll** (the .NET-side stubs) and **#jdglue.dll**(the glue between the two). Those components can be found in the usual LISA locations (the **bin** and **lib** directories).

The easiest way to take advantage of it is using the custom JavaScript step, but extensions would be possible also. We will cover only the Java -> .NET API here because it is the natural usage from LISA. There are only three central classes:

com.itko.lisa.jdbridge.JDInvoker

```
/** Loads the .NET CLR in the Java process */
public static native void startCLR();

/** Stops and unloads the .NET CLR from the Java process */
public static native void stopCLR();

/**
 * Invokes a methods in the specified .NET assembly (.dll or .exe).
 * @param assembly the full path to the assembly the type resides in
 * @param type     the fully qualified name of the type on which to invoke
 * @param method   the name of the method to invoke
 * @param args     an array of arguments expected by the method
 * @return the return value of the .NET method
 */
public static Object invoke(String assembly, String type, String method, Object ... args)
```

com.itko.lisa.jdbridge.JDProxy

```
/**
 * Returns a proxy to a .NET instance that exists in the CLR after invoking its constructor.
 * @param assembly the full path to the assembly the type resides in
 * @param type     the type to instantiate
 * @param args     an array of arguments expected by the constructor
 * @return a proxy to the .NET instantiated type
 */
public static JDProxy newInstance(String assembly, String type, Object ... args)

/**
 * Invokes the specified method on the object represented by this proxy
 * @param method the name of the method to invoke
 * @param args   an array of arguments expected by the method
 * @return the return value of the method
 */
public Object invoke(String method, Object ... args)

/**
 * If the object represented by this proxy exposes .NET event delegates, this method enables the
Java
 * program to register event listeners in Java code.
 * @param event the name of the event to listen for
 * @param l     the listener interface whose onEvent method gets invoked when the event is fired.
 */
public void addListener(String event, JDProxyEventListener l)

/**
 * Removes an event listener previously added via addListener.
 * @param event the name of the event to listen for
 * @param l     the listener interface whose onEvent method gets invoked when the event is fired.
 */
public void removeListener(String event, JDProxyEventListener l)

/**
 * Method to invoke to release resources when done with the proxy.
 */
public void destroy()
```

com.itko.lisa.jdbridge.JDProxyEventListener

```
/**
 * This method gets invoked (from .NET) on all listeners registered via JDProxy's addListener
method.
 * @param source the proxy to the object raising the event (on which addListener was called)
 * @param evt    the name of the event being raised
 * @param arg    a string representation of event data
 */
public void onEvent(JDProxy source, String evt, Object arg)
```

As usual when two technologies communicate with each other, it is important to understand the marshaling mechanism for arguments and return values. The approach taken by jdbridge is similar to RMI and .NET remoting; that is, there is marshaling by value or by reference.

All primitive types (Boolean, byte, short, char, int, long, float, double), and Strings are marshaled by value and map one to one between Java and .NET so no special handling is required.

Similarly, framework collections classes are mapped one to one (Java's **Lists** to .NET's **Lists** and Java's **Maps** to .NET's **Dictionaries**).

For general objects, if the .NET object implements the **IXmlSerializable** interface, it is marshaled back to Java by value, which means that a Java class with the exact same format (package, name, methods, and so on) must exist in the classpath; otherwise it is marshaled by reference as a **JDProxy**. This lets you chain **JDProxy** calls and pass a **JDProxy** as an argument to another **JDProxy** call.

Exceptions raised in .NET are also propagated to Java and thrown as **RuntimeExceptions** with a stack trace spanning both Java and .NET code.

Example

Here is a simple example. Say there is a .NET dll named **AcmeUtils.dll** in LISA's bin directory. This dll contains the type **com.acme.Calculator** that has all the usual arithmetic functions, **Add, Subtract**, and so on, and you want to invoke those from a JavaScript step. Here is a script that does this:

```
import com.itko.lisa.jdbridge.JDInvoker;
import com.itko.lisa.jdbridge.JDProxy;

JDInvoker.startCLR();
JDProxy calc = JDProxy.newInstance(Environment.LISA_HOME + "bin\\AcmeUtils.dll",
"com.acme.Calculator", new Object[0]);

Integer sum = (Integer) calc.invoke("Add", new Object[] { 3, 4 });
Double ratio = (Double) calc.invoke("Divide", new Object[] { 3.0, 4.0 });
Integer square = (Integer) calc.invoke("Square", new Object[] { 5 });
...
```

The arguments are passed explicitly into an array of Objects. This is because LISA's BeanShell does not support the varargs (...) notation. If you were to code this in an extension, the syntax becomes less cumbersome.

Now let us say the **Calculator** object exposes the **OnCalculationStart** and **OnCalculationEnd** events and we wanted to subscribe to those so as to measure the duration of the calculation:

```
...
JDProxy calc = JDProxy.newInstance(Environment.LISA_HOME + "bin\\AcmeUtils.dll",
"com.acme.Calculator", new Object[0]);

calc.addListener("OnCalculationStart", new JDProxyEventListener() {
    public void onEvent(JDProxy source, String evt, Object arg) {
        //capture timestamp here
    }
});

calc.addListener("OnCalculationEnd", new JDProxyEventListener() {
    public void onEvent(JDProxy source, String evt, Object arg) {
        //capture timestamp here
    }
});

Long fact = (Long) calc.invoke("Factorial", new Object[] { 30 });
...
// diff the timestamps here
...
```

Finally, if your project involves extensive use of .NET assemblies it would probably be a good idea to wrap all interactions with .NET code inside compiled extensions. In this case the standard pattern would look like this:

```
package com.acme;

import com.itko.lisa.jdbridge.JDInvoker;
import com.itko.lisa.jdbridge.JDProxy;
import com.itko.lisa.jdbridge.JDProxyEventListener;

public class Calculator {
    static {
        JDInvoker.startCLR();
    }

    private JDProxy m_proxy = JDProxy.newInstance(Environment.LISA_HOME + "bin\\AcmeUtils.dll",
"com.acme.Calculator");

    public int add(int x, int y) {
        return ((Integer) m_proxy.invoke("Add", new Object[] { x, y })).intValue();
    }

    ...
}
```

and this could trivially be invoked from a custom JavaScript step or even the Complex Object Editor (COE).