CA LISA

Developer Guide

Version 7.5.1

Contact CA Technologies

Contact CA Support

For your convenience, CA Technologies provides one site where you can access the information that you need for your Home Office, Small Business, and Enterprise CA Technologies products. At http://ca.com/support, you can access the following resources:

- Online and telephone contact information for technical assistance and customer services
- Information about user communities and forums
- Product and documentation downloads
- CA Support policies and guidelines
- Other helpful resources appropriate for your product

Providing Feedback About Product Documentation

If you have comments or questions about CA Technologies product documentation, you can send a message to techpubs@ca.com.

To provide feedback about CA Technologies product documentation, complete our short customer survey which is available on the CA Support website at http://ca.com/docs.

CA LISA Fix Strategy

CA LISA Release 7.5 Fix Strategy

Installation and Maintenance Methodology

CA LISA uses Install4J to install the base product. We deliver fixes in the form of individual Java JARs that you add to the **$LISA_HOME/bin/patches** directory.

Maintenance Delivery and Time Frames

CA LISA provides maintenance using the previously mentioned delivery methods. The goal of the CA LISA product team is to generate these deliveries using the following guidelines:

- **Individual Fixes:** As necessary
- **Maintenance Releases:** Every 3 to 4 months

Maintenance releases are provided as new, full installations that mirror the GA release.

For the GA-1 version of CA LISA, maintenance is provided on an as-necessary basis.

Note: In accordance with the CA Corporate Support Policy on the CA Support web page, CA defines supported releases as (A) the latest major generally available (GA) release and (B) one previous major release (GA-1).

For the current CA LISA product line, CA LISA 7.5 represents the GA version and CA LISA 7.1 represents the GA-1 release.

Delivery Definitions

Individual Fix

A published fix, which addresses the product defect as documented in the original test fix. An individual fix can contain binary patches, or one or more cumulative replacement components or elements, and must be applied to an existing product environment. The individual fixes are included in and superseded by the next applicable cumulative fix or cumulative release.

Maintenance Release

A release of the product that does not add new features or content. A maintenance release contains an accumulation of fixes from the GA release to a given point in time.

Contents

Chapter 1: Developer Guide Overview

LISA functionality allows for a high amount of customization. Programmers can extend this functionality through the Software Development Kit (SDK).

Note: The SDK was previously known as the LISA Extension Kit (LEK).

This section contains the following topics:

Examples and API Documentation

The examples_src.zip file contains the examples that are referenced in this guide. This zip file is included with the LISA installer and is located in the LISA_HOME\examples_src directory.

The SDK JavaDocs file contains the API documentation. The SDK JavaDocs are available from the Reference Documentation section of the CA LISA Bookshelf. You can download the JavaDocs as a zip file or view them in an HTML format.

Chapter 2: The Integration API

LISA provides an integration Application Programmer Interface (API) that lets you manage LISA test execution within Java-based server-side components. This chapter details the basic concepts central to the LISA integration API.

This section contains the following topics:

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 about 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:** The 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 (see page 15).
- **TransInfo Class:** Abbreviation for Transaction Information. The integrated component uses this class to indicate testing events, such as a test failure or success. For more information about using the TransInfo class, see Integrating Components (see page 15).
- **HasLisaIntegrator Interface:** Implemented by object types that are returned by methods that are integrated using Java-based integrators. These object types provide 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 (see page 21).
- **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 Integration Filters (see page 15).
- **Integration Assertions:** Special assertion elements that take advantage of the additional information that integrated server-side components provide. For example, a Check Integrator Response (see page 26) assertion can set the next test step based on the build status reported by the integrated component. For more information about integration assertions, see Integration Assertions (see page 15).

Integration Flow

The integration flow includes the following steps.

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 the communication with the running LISA test case. 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 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 Integrate Server-Side Components (see page 16).
3. Handle the output from the integrated method.
   For more information about handling integrated output, see Handle Integrated Output (see page 21).
4. Use integration filters in the test case that tests the server-side component.

   For more information about integration filters, see Integration Filters (see page 15).
5. Use integration assertions in the test case that tests the server-side component.

   For more information about integration Assertions, see Integration Assertions (see page 15).

Chapter 3: Integrating Components

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

This section contains the following topics:

Integrate Server-Side Components

This procedure describes how to LISA-enable a server-side component.

Follow these steps:

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 must happen in context of a transaction. Therefore, before beginning to communicate, start a transaction using the application-specific integrator. You can ensure 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 method is a useful mechanism for getting information out of an integrated component, especially when used 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 LISA property named 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, and 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 and passes the string message that is logged.

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

  For more information about TransInfo assertion methods, see Assertion Methods (see page 19).

- Force the next node. On rare occasions, it is 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 (see page 21).

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 failed. You can separate out parts of the transaction using subcomponents and can report their individual statuses. The **com.itko.lisaint.CompInfo** class represents a subcomponent.

For example, the following code creates a subcomponent of the transaction and sets its build status to STATUS_SUCCESS:

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

Subcomponents are treated as part of the overall transaction. If one subcomponent fails, the entire transaction fails. The following code creates a subcomponent 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.

Collect Transaction Information

The TransInfo class encapsulates all the information that is gathered about the execution of a specific "transaction" of the system under test. This class 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 subcomponents 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. The 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 as 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 a 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 the 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 Collect Transaction Information (see page 19).

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

Handle 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 described wrapping 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 returns value types or the class you are executing itself implements 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 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.

Chapter 4: Testing Integrated Components

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

This section contains the following topics:

Integration Filters

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

- The LISA software must 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 subcomponent 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 specified attributes, then execute the node that was specified.
- **Report Component Content:** If this box is selected, the servlet application tested includes the actual content of the components in the output.

Note: 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 can ignore the request and not provide the component content.

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

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 Integrator Response (see page 26) 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 (see page 26)
- Check LISA Integrator Component Content Response (see page 27)
- Check LISA Integrator Reporting Missing Data (see page 27)

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

Check 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 subcomponent 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 *User Guide*.

Check 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, regardless of the expression.
  - An empty expression is never a hit.
  - A * matches any not null value.
  - Any other expression is considered a regular expression.

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

Check 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. The expression is evaluated in the following way:
  - An empty expression is never a hit.
  - A * matches any not null value.
  - Any other expression is considered a regular expression.

Note: 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 *User Guide*.

Chapter 5: Extending LISA Software

This chapter explains the main concepts that are involved in extending the LISA software.

This section contains the following topics:

Reasons to Extend the LISA Software

The LISA software provides most of the elements that are required to test enterprise software components. However, you can 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 could create the following:

- Custom node to test the transfer of files
- Custom node to verify the contents of transferred files
- 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 File

The **lisaextensions** file is where you define the extension points for LISA.

One or more custom extensions (filters, assertions, test steps, and reports, for example) are declared in a file whose extension is .lisaextensions. The name must be unique relative to any other lisaextensions files in the same environment. For example, there could be two lisaextensions files with the names **library-A.lisaextensions** and **library-B.lisaextensions**.

The lisaextensions file must be included in a JAR with the classes that do the custom extension implementation. The JAR must be placed in the LISA 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.

Note: Using the **typemap.properties** file to define extensions is supported only for backward compatibility.

All custom extensions in LISA must provide a controller, viewer (frequently called an editor), 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. You can create custom versions if these defaults are not suitable for your needs.

Use of the lisaextensions file

The custom element is 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 on the right side of the equal sign represents the location of the classes that implement the appropriate node logic. If more than one assertion is defined, all corresponding classes are mentioned in the same line, separated by commas, as in:

```
asserts=com.mycompany.lisa.AssertFileStartsWith,com.mycompany.lisa.AnotherAss
ert
```

Controller and Editor Objects

LISA performs a lookup on each element that is declared as an implementation object to get the controller and the editor to associate with that implementation. For example, the assertion in the previous example has the following extra entry in the lisaextensions file:

```
com.mycompany.lisa.AssertFileStartsWith=com.itko.lisa.editor.DefaultAssertCon
troller,com.itko.lisa.editor.DefaultAssertEditor
```

The format of this properties file entry is

```
implementation_class_name=controller_class_name,editor_class_name
```

In this example, the default controller and editor are used. The defaults are adequate for most cases. However, you can encounter situations in which custom controllers or, more frequently, custom editors must be provided. A lisaextensions file is the place to declare the classes corresponding to them.

wizards.xml File

The **LISA_HOME** home directory 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. You can add your custom LISA extensions to this file to make them easily available to users. You can also remove built-in elements that are not used frequently. The **wizards.xml** file itself provides some useful information about how to read and modify its contents, but the concepts are 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. For example:

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

LISA predefines the element values (in this case "assert") and the types (in this case "http"). New element values or types are ignored. The type is the short name for the kind of elements that is described in the contained Entry tags. In this example, the Entry tags are 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:

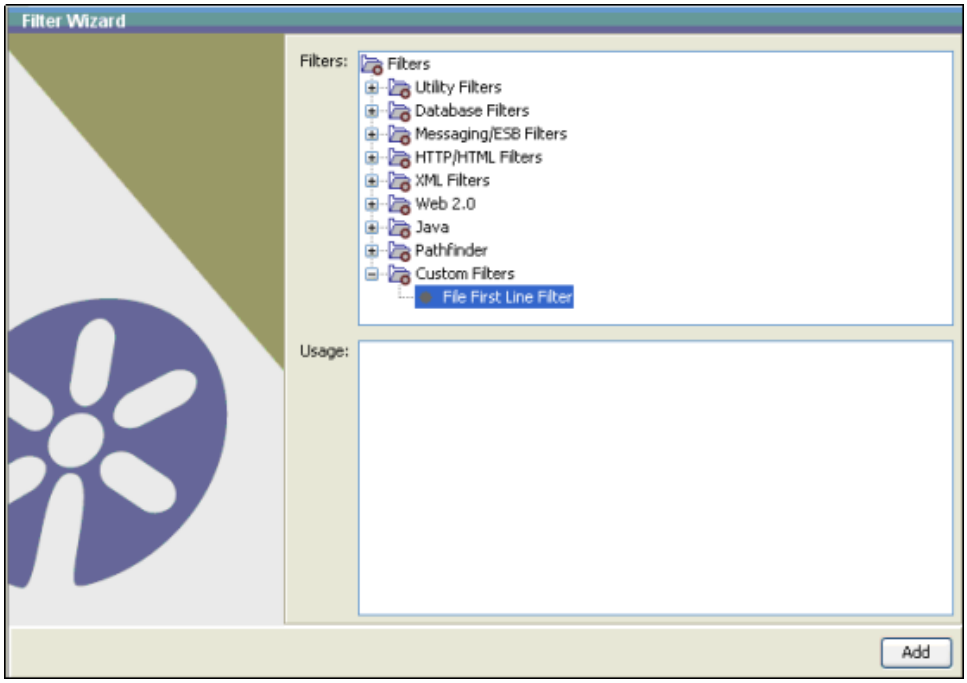
```
<Entry>
   <Name>Make Diff-like Comparisons on the HTML Result</Name>
   <Help>This is the right assertion to use when you have a HTML result and you
wish to perform diff-like operations on it. It lets you define simply what portions
of the text may change, what may not change, and what portions 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:** The label that is used in the wizard itself, shown as an 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 is shown when this entry has been selected.
- **Type:** 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 that is associated with a LISA object viewer. Every class that 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 that is associated with the element in the model editor. For example, notice the text "File First Line Filter" in the following image.



A method in the class that defines the filter provides the text. 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 that the element exposes 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 **com.itko.lisa.test.Parameter**. The previous example 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 that is 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 that is associated with the parameter. For example, passing **Boolean.class** renders the parameter as a check box. Passing **TestExec.PROPERTY_PARAM_TYPE** renders the parameter as a drop-down list 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 Test Exec 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 that the TestExec class provides 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:** Indicates 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:** Indicates 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.

Chapter 6: Extending Test Steps

The test steps that LISA provides include most of the logic that is required to test enterprise software. However, you can create your own test step to accommodate a specific situation.

Each existing test step provides a step-specific Swing user interface to help users develop that type of test step. You can provide this same support by creating a Java class that extends **com.itko.lisa.test.TestNode** and providing a Swing user interface. However, there is a much simpler way to provide a custom test step.

A set of name-value pairs can adequately represent many testing situations. 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 that is required to test. You need only:

- The FTP host
- The full path and name of the file
- The username and password that is used to access the FTP host

A name-value pair can represent each of these values.

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

The LISA SDK provides two ways to enable you 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 that developers use.
- **Native Test Steps:** These test steps are created in precisely the way that ITKO develops test steps within LISA and can appear in their own categories.

This section contains the following topics:

Custom Java Test Steps

This section explains how to create a custom Java test step and use it in the model editor.

This type of test step is created by extending an abstract base class that LISA provides in the **Lisa.jar**. A custom Java test step is faster and easier than the native test step because it does not require you to build a user interface 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 in an appropriate manner, but some parameters could require a customized editor. For those needs, consider creating a native test step.

Topics include:

Create a Custom Java Test Step

Follow these steps:

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 method is 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 that are specified here are used as the 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 an anonymous login. Limit unnecessary hits on the SuSE FTP site.

4. Implement the required **executeNodeLogic** method.

   This method defines the logic that runs when the test step executes. Typically, this method 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;
}
```

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

- It encourages proper resource handling. Do not pass common resources, such as files and JDBC connections, as results. Doing so prevents the node performing a cleanup by calling a close method. If you construct your own data object, you can store and pass only 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 about using data objects to perform conditional filtering, see Create a New Filter (see page 53).

The previous code uses the **FileDataObject** custom data object to store only 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**.

Deploy a Custom Java Test Step

You must make a custom Java test step available to the model editor before you can use it in a test case.

To deploy a custom Java test step:

1. Create a lisaextensions file or use an existing one if one exists.

   In the lisaextensions file, mention this class against the 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_HOME/hotDeploy** directory.

   If your custom Java test step depends on any third-party libraries, copy those libraries to the **LISA_HOME/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 that is packaged at **LISA_HOME/doc/DevGuide/lib/ftp.jar**.

   Copy both of these files to the **LISA_HOME/hotDeploy** directory.

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

   This client puts the given class name in a convenient drop-down list for users when authoring a test case. This 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, exit and restart the program for this new setting to take effect.

Define a Custom Java Test Step

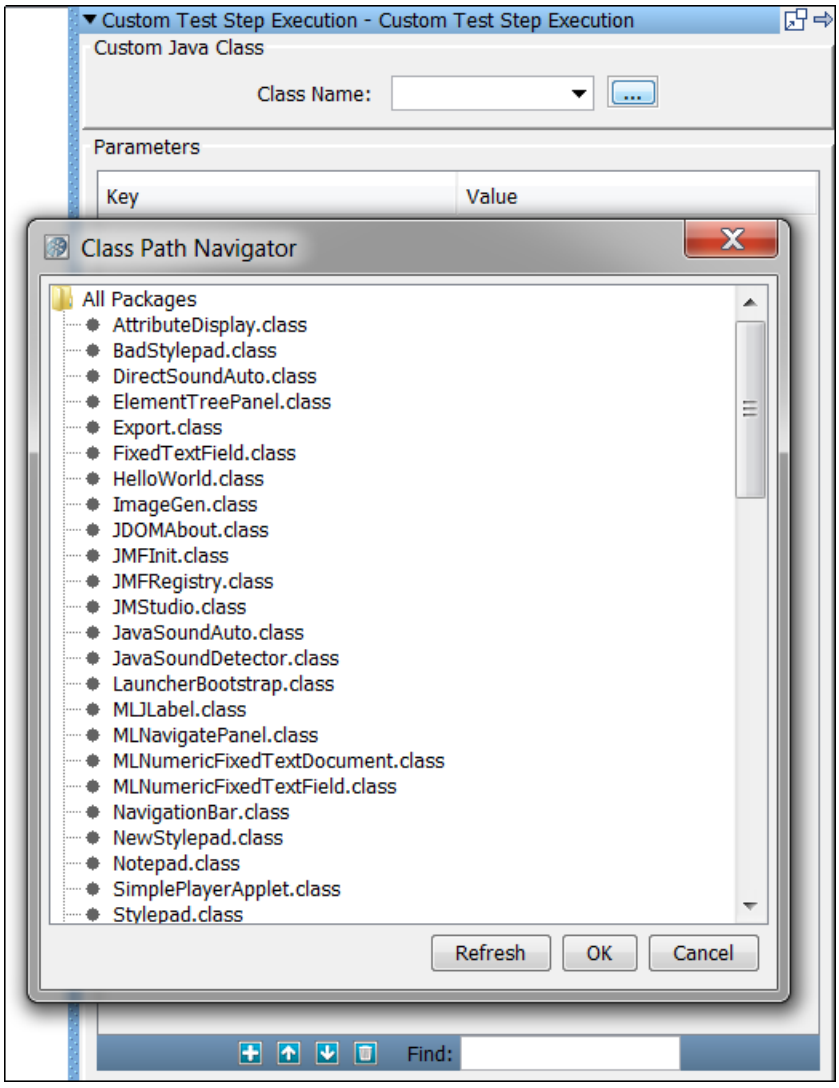
This procedure describes defining a custom Java test step in the model editor.

Follow these steps:

1. Change the **Type** of the node to **Custom Test Step Execution**.
2. Specify the custom test step to test.

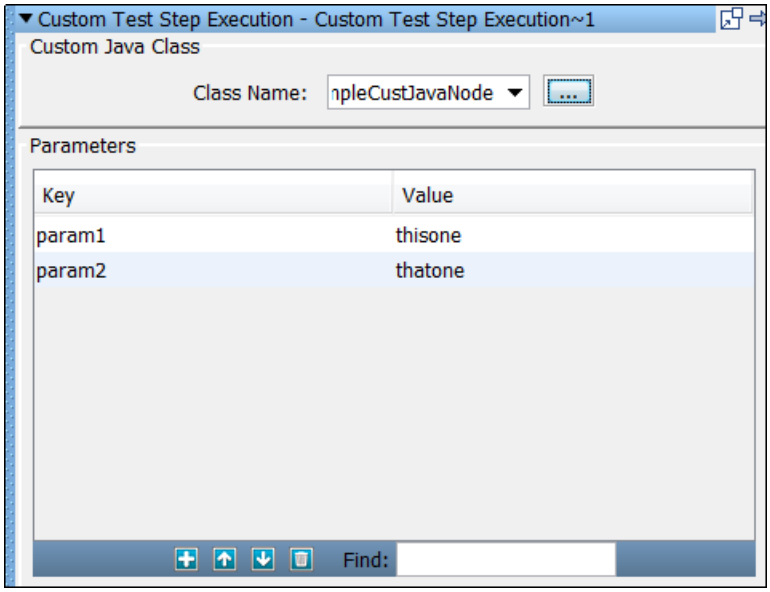
   Enter or select the fully qualified name of the Java class in the Class Name field.

   To specify a custom Java test step class easily, click the ellipsis (...) next to the Class Name field to launch the Class Path Navigator. The Path Navigator only looks for custom test steps. Select the test step class that you want to test and click OK.



3. Set the parameters to the custom test step.

   For each parameter in the Parameters list, supply a value.



Native Test Steps

This section explains how to create a native test step and use it in the model editor. The native test step is the way to provide 100 percent customized test step authoring and execution for your test steps. In this test step, you are responsible for providing the editing environment and the complete execution engine for your test step. This process is precisely how native test steps provided as built-in within LISA are developed.

Topics include:

- Create a Native Test Step (see page 44)
- Deploy a Native Test Step (see page 45)
- Define a Native Test Step (see page 45)

Create a Native Test Step

This procedure describes creating a native test step.

Follow these steps:

1. Create a Java class that extends **com.itko.lisa.test.TestNode**.

   This class implements the runtime logic of the test step. See the Javadoc for **TestNode** and the sample code that is provided for **FTPTestNode** in the samples that accompany this document. The following calls are important for the 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 the operation from the test case XML DOM. As the test case is constructed, it is passed with the DOM Element of the 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 is invoked. The LISA 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 various tasks. See the Javadoc on **TestExec** and the description of this class in Integrating Components (see page 15).

2. Create a Java class that extends **com.itko.lisa.editor.TestNodeInfo**.

   To create and edit your node, provide a controller and viewer in the MVC pattern. The **TestNodeInfo** class is the base class for all test steps that are developed for LISA. The authoring framework executes this class to interact with your test step data and logic. See the Javadoc for **TestNodeInfo** and review the example class that is provided, named **FTPTestNodeController**.

3. Create a Java class that extends **com.itko.lisa.editor.CustomEditor**.

   This class provides LISA with the actual user interface for viewing and editing your node data. The authoring framework constructs and calls this class to display your parameters, verify their validity, and save your changes back into the **TestNodeInfo** extension. 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.

4. Perform the following method overrides if you want to control how the default name of the step is generated.

   - Override the **generateName** method in the class that extends **TestNode**. In this method, add the logic to create the default name. The following example is for the **FTPTestNode** sample class. When the user creates an **FTP** custom step, the default name consists of FTP followed by the file name.

     ```
     public String generateName()
     {
         return "FTP " + getFile();
     }
     ```

- Override the **generateName** method in the class that extends **TestNodeInfo**. In this method, add a call to the **generateName** method in the class that extends **TestNode**. The following example is for the **FTPTestNodeController** sample class.

```
public String generateName()
{
    FTPTestNode ftp = (FTPTestNode)getAttribute( FTP_KEY );
    return ftp.generateName()
}
```

Deploy a Native Test Step

Native test steps must be explicitly declared to LISA at startup of the Test Manager. This declaration allows the authoring framework to associate the three classes that were 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. This example is used in the tree of the user interface. 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.FTPTestNodeContro
ller,com.mycompany.lisa.node.FTPTestNodeEditor
```

For more information about the lisaextensions file, see Extending the LISA Software (see page 29) in the *Developer Guide.*

Define a Native Test Step

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

Chapter 7: Extending Assertions

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

This section contains the following topics:

Create a New Assertion

The assertions that the LISA software provides contain most of the logic that is required to test enterprise software. However, you can create your own assertion to handle a specific situation.

LISA provides built-in support for custom assertions. To create a custom assertion, you create a Java class that extends **com.itko.lisa.test.CheckResult**. This class handles most of the behind-the-scenes logic that is required to execute the assertion, and provides a nice user interface in the model editor.

Follow these steps:

1. Create a Java class that extends **com.itko.lisa.test.CheckResult**.

   This class 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 method provides the name that is used to identify the custom assertion in the model editor.

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

   The string that the **getTypeName** method returns is the default name of a new assertion.

3. Handle custom parameters to the assertion.

   Some assertions require more information than is provided in the node execution result. In this case, provide a parameter that allows the user to specify that information in the Assertions tab of the model editor.

   Implement the abstract **getCustomParameters** method. In this method, you create a **ParameterList** and add a **Parameter** for each parameter to the assertion. The constructor for the Parameter takes the following:

   - 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. It 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 can include an assertion that looks like the following example:

```
<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 must 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 ensure that 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 prefer to read from the DOM Element. 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 results returned from executing the node. The Boolean return type returns **true** if the assertion is true. Otherwise it returns **false**.

The following code verifies that the first line of the file that is passed from the node contains the string that is stored in the **param** parameter. This code 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, simply declare whether the assertion as defined is true or false. Do not worry about the steps to execute after the state is returned. The LISA workflow engine evaluates whether that is considered a fired assertion or not.

Deploy a New Assertion

You must make a custom assertion available to the model editor before you can use it in a LISA test case.

To deploy a custom assertion:

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

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

   com.mycompany.lisa.AssertFileStartsWith=com.itko.lisa.editor.DefaultAssertCon
   troller,com.itko.lisa.editor.DefaultAssertEditor
   ```

   You can 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_HOME/hotDeploy** directory.

   If your custom assertion depends on any third-party libraries, copy those libraries to the **LISA_HOME/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, exit and restart the program for this new setting to take effect.

Define and Teste a New Assertion

This procedure describes defining a custom assertion.

Follow these steps:

1. Change the Type of the assertion, selecting the text that you specified in the **getTypeName** method.
2. Set the 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.

Chapter 8: Extending Filters

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

This section contains the following topics:

Create a New Filter

The filters that the LISA software provides includes most of the logic that is required to test enterprise software. However, you can create your own filter to handle a specific situation. The LISA software provides built-in support for custom filters.

Follow these steps:

1. Create a Java class that implements **com.itko.lisa.test.FilterInterface**.

   This class tells the LISA software that your class is a custom filter.

   ```
   public class FilterFileFirstLine implements FilterInterface
   {


   }
   ```

2. Implement the required **getTypeName** method.

   This method provides the name that is used to identify the custom filter in the model editor.

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

3. Define the parameters to the filter.

   For each item in the Filter Attributes section of the Filters tab in the model editor, add a **Parameter** to the **ParameterList** for the filter. In this example, the custom filter takes two parameters:

   - A check box to identify whether 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. LISA 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 can include a filter that looks like the following example:

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

In the **initialize** method, set the **isftp** variable to **true** and set 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**. This class finds child tags of the given parent tag and reads 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 **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.

If a filter defines the **prop** key, then the default name of the filter is **{{propValue}}. propValue** is the value of the **prop** parameter. If a filter does not define the **prop** key, then the default name of the filter is the string that the **getTypeName** method returned. The FilterFileFirstLine sample class defines this key as:

```
static private String PROP_PARAM = "prop";
```

5. Because the filters execute before and after the test step, you get two chances to implement the filter logic.

Implement the filter logic before node execution with the **preFilter** method. The **TestExec** parameter provides access to the test environment, such as logs and events. If the filter set a new node to execute, the Boolean return type returns **true**. Otherwise, it returns **false**.

In this example, we are not interested in filtering before the node executes, so the **preFilter** method does nothing.

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

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

In this example, we store the first line of the file that is returned from the node in the new property. ftp:// is prepended if the **isFTP** box is selected.

A LISA user can use a filter at either the test step level or the test case level. Add logic to the filter that verifies whether the result is the proper state to run the filter. For example, if your filter assumes that the LASTRESPONSE holds a FileDataObject, then verify 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 this filter can reference. For example, a filter that would set the next node to "fail" on a given condition would make a NodeConnection object to encapsulate that reference. See **com.itko.lisa.test.NodeConnection** in the Javadoc for more on this class. This method is also how test elements are informed when the name of a test step is changed.

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

Deploy a New Filter

You must make a custom filter available in the model editor before you can use it in a test case.

Follow these steps:

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.DefaultFilterEditor
   ```

   You can also add this filter to the wizards in the **wizards.xml** file.

2. Copy the JAR file that contains your custom filter and **lisaextensions** file to the **LISA_HOME/hotDeploy** directory.

   If your custom filter depends on any third-party libraries, copy those libraries to the **LISA_HOME/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, exit and restart the program for this new setting to take effect.

Define and Test a New Filter

This procedure describes defining a new LISA filter.

Follow these steps:

1. Change the Type of the filter, selecting the text that you specified in the **getTypeName** method.
2. Set the parameters to the custom filter.

   For each parameter in the **Filter Attributes** section, supply a value.
3. Test a custom filter like you would any built-in filter.

Chapter 9: Custom Reports

This chapter explains how to extend LISA with new reports.

This section contains the following topics:

Create a New Report Generator

The report generators that LISA provide include most of the output required. However, you can create your own report to handle a specific situation. LISA provides built-in support for custom report generators.

To create a report generator:

1. Create a Java class that extends **com.itko.lisa.coordinator.ReportGenerator**.

   This class tells LISA that your class is a custom report.

   ```
   public class ReportEventsToFile extends ReportGenerator
   {


   }
   ```

2. Implement the required **getTypeName** method.

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

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

3. Define the parameters to the report.

   For each item in the **Report Attributes** section of the **Reports** tab in the Staging Document Editor. Add a **Parameter** to the **ParameterList** for the report.

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. It 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 **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 invokes the pushEvent method for every event you have not filtered.

6. Implement the **finished** method.

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

Deploy a New Report Generator

You must make a custom report available before you can use it in a staging document.

Follow these steps:

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

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

   You can also add the report through the **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_HOME/hotDeploy** directory.

   If your custom report depends on any third-party libraries, copy those libraries to the **LISA_HOME/hotDeploy** directory.

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

Use a New Report Generator

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

Chapter 10: Custom Report Metrics

This chapter explains how to extend the LISA software with a new reporting metric.

This section contains the following topics:

Create a New Report Metric

When a test case is staged, a subsystem within LISA samples metric values. LISA then reports them in the ways that are defined in the staging document. The staging document includes the metrics to be collected. Users can also add them as the test runs.

Two classes must be created for the 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.

Follow these steps:

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 method that provides the name that is 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 be collected.

4. Create a Java class that extends **com.itko.lisa.stats.MetricCollector**.

   LISA calls on instances of this class to collect the requested metrics. 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.

Deploy a New Report Metric

You must make a custom report metric available before you can use it in a staging document.

Follow these steps:

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

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

   You can also add the report metric through the **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 libraries to the **LISA hotDeploy** directory.

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

Chapter 11: Custom Companions

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

You can create custom companions for LISA in two different ways. Native companions are created in much the same way that Native test steps are created. And somewhat like the Custom Java test step, there is a simpler way in which LISA shields you from most of the editor and serialization overhead. That approach is documented here.

Note: If a companion contains duplicate parameter values, the duplicates are filtered out when you save the test case. You must close the test case and then reopen it to see the change in the user interface. This problem is specific to custom companions because of an issue with the implementation of the lifecycle of custom companions.

If you have a test case containing a custom companion that wrote duplicate parameters into the .tst file, there is a workaround. Reopen the .tst file, click save, close the .tst file, and then reopen it. The duplicates are removed during the save, but the companion UI does not update to reflect this change. The close and reopen step is required to see the change.

This section contains the following topics:

Create a New Companion

Follow these steps:

1. Create a Java class that extends **com.itko.lisa.test.SimpleCompanion**.

   This class provides LISA all the information that is required to create, edit, and execute your companion logic.

   ```
   public class AllowedExecDaysCompanion extends SimpleCompanion implements
   Serializable
   {

   }
   ```

   Ensure that you implement Serializable. This is important when your companion is used in remotely staged tests.

2. Implement the required **getTypeName** method.

   This method provides the name that is used to identify the companion in the model editor.

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

3. Implement the **getParameters** method.

   This method that provides LISA with the parameters you need for your companion to be executed. Also call the superclass implementation of this method. The model editor allows you to edit the parameters that are shown here. These parameters are given to you at the time of execution. You do not have to implement this method if your companion does not require 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;
   }
   ```

4. Implement the **testStarting** method.

   LISA calls this method with the parameters you requested. If an error occurs or you otherwise want to prevent the test from executing normally, throw a **TestRunException**.

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

5. Implement the **testEnded** method.

LISA calls this method 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
```

Deploy a New Companion

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

Like the Native Test test step, three classes are required for companions, but LISA provides default implementations for the two classes that are not documented here. The default controller is **com.itko.lisa.editor.CompanionController** and the editor is named **com.itko.lisa.editor.SimpleCompanionEditor**. Notice that the built-in LISA companion named **Set Final Step to Execute Companion** is defined with these classes. Use the registration for that companion as a sample. They are connected in the **lisaextensions** file.

```
companions=com.mycompany.lisa.AllowedExecDaysCompanion
com.mycompany.lisa.AllowedExecDaysCompanion=com.itko.lisa.editor.CompanionCon
troller,com.itko.lisa.editor.SimpleCompanionEditor
```

For more information about the **lisaextensions** file, see Extending the LISA Software (see page 29).

As with all custom LISA test elements, you must make the classes that you have developed and the **lisaextensions** file available to LISA. The most common way to do make them available is to put a JAR file in the **LISA_HOME/lib** directory.

Note: A custom companion can implement the **StepNameChangeListener** interface and can receive notification when the name of any step is changed. An SDK developer implements 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 select the teardown step. This code change was added to enable notifying the **Final Step to Execute** companion when a step name changes.

Chapter 12: Using Hooks

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

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

Hooks are used as follows:

- To configure test environments.
- To prevent tests that are not properly configured or do not follow defined best practices from executing.
- To provide common operations.

Anything that a hook can perform can also be modeled as a companion in LISA. However, there are several 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 want to prevent users from accidentally not including it, a hook is preferable.
- Companions can have custom parameters and are rendered in the model editor. Hooks are practically invisible to the user and therefore can request no special parameters. Hooks 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. Assume a test is run on two computers. One computer has a hook that is registered, and the other does not. The hook runs only when the test is staged on the computer where it is explicitly deployed. The defined companions execute regardless of any install-level configuration.

This section contains the following topics:

Create a New Hook

Follow these steps:

1. Create a Java class that extends **com.itko.lisa.test.Hook**.

   This class gives LISA all the information that is required to execute the logic for your hook.

   ```
   public class HeadlineHook extends Hook
   {

   }
   ```

2. Implement the **startupHook** method.

   LISA calls this method when the test starts. If an error occurs or you otherwise want 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 )
   ```

Deploy a New Hook

Hooks are deployed when a class name in the **lisa.properties** file registers them. 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.

Chapter 13: Custom Data Sets

This chapter explains how to extend LISA with a new data set.

Data sets are created in much the same way that test steps are created. LISA provides a simpler way that shields you from most of the editor and serialization overhead. This approach uses default class implementations for the controller and editor. This simpler approach is documented here.

This section contains the following topics:

Data Set Characteristics

Data sets are different from 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.

LISA provides the infrastructure for remoting your custom data set automatically. There are typically no specific issues that are 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 do not have access to a **TestCase** or **TestExec** object. Your class is 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.

Create a New Data Set

Follow these steps:

1. Create a Java class that extends **com.itko.lisa.test.DataSetImpl**.

   This class gives LISA all the information that is required to execute your data set logic.

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

   Your data set object is a Remote RMI object, and is therefore able to throw a **RemoteException** from its constructor and some methods.

2. Implement the required **getTypeName** method.

   This method provides the name that is used to identify the companion in the LISA Test Case Editor.

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

   The string that the **getTypeName** method returns is the default name of a new data set.

3. Implement the **getParameters** method.

   This method provides LISA with the parameters you need for your companion to be executed. You must also call the super class implementation of this method. The LISA Test Case Editor allows you to edit the parameters that are shown here. The parameters are given to you at the time of execution. You do not have to implement this method if your companion does not require parameters.

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

   For more information about Parameters and ParameterLists, see Extending the LISA Software (see page 29).

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.

   LISA calls this method when another row is needed from the data source for the data set. If an error occurs or you otherwise want 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 wants:

   - Restart reading the data source from the top again automatically, or
   - Return a null from this method to indicate the data source is out of rows so that the test workflow reflects the condition.

   The following example shows possible psuedo-code for this function:

   ```
   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
   ```

Note: The API for the DataSet interface includes the public **String getType()** method. This method returns the classname of the class implementing this interface.

Deploy a New Data Set

Data sets must be explicitly declared to LISA at startup of 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. LISA provides default implementations for the two classes that are not documented here. The default controller is **com.itko.lisa.editor.DataSetController** and the editor is named **com.itko.lisa.editor.DefaultDataSetEditor**. Notice that some of LISA's built-in data sets use these classes.

They are connected in the **lisaextensions** file, as follows:

```
datasets=com.itko.examples.dataset.CSVDataSet

com.itko.examples.dataset.CSVDataSet=com.itko.lisa.editor.DataSetController,c
om.itko.lisa.editor.DefaultDataSetEditor
```

See the information about the **lisaextensions** file in Extending the LISA Software (see page 29) for more details on how to register your data set.

As with all custom LISA test elements, make the classes that you have developed and the **lisaextensions** file available to LISA. The most common way to make them available is to bundle them in a jar file that is placed in the LISA hot deploy directory.

Chapter 14: 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 **b**ridge) and is made of three components:

- jdbridge.jar (the Java-side stubs)
- djbridge.dll (the .NET-side stubs)
- #jdglue.dll(the glue between the two)

These components can be found in the usual LISA locations (the bin and lib directories).

The easiest way to take advantage of this bridge is by using the custom JavaScript step, but extensions are also possible. This section covers only the Java -> .NET API because it is the natural usage from LISA. The following classes are the three central classes:

- com.itko.lisa.jdbridge.JDInvoker
- com.itko.lisa.jdbridge.JDProxy
- com.itko.lisa.jdbridge.JDProxyEventListener

This section contains the following topics:

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 that jdbridge takes is similar to RMI and .NET remoting in that 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. No special handling is required.

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

For general objects, if the .NET object implements the **IXmlSerializable** interface, it is marshaled back to Java by value. This 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 that are 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. Assume there is a .NET dll named **AcmeUtils.dll** in the LISA bin directory. This dll contains the type **com.acme.Calculator** that has all the usual arithmetic functions: **Add**, **Subtract**, and so on. You want to invoke those functions from a JavaScript step. Here is a script that does invokes them:

```
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 because the LISA BeanShell does not support the varargs (...) notation. If you were to code this in an extension, the syntax becomes less cumbersome.

Suppose the **Calculator** object exposes the **OnCalculationStart** and **OnCalculationEnd** events and you want to subscribe to those events 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
...
```

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

```
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();
    }

    ...
}
```

This extension can be invoked from a custom JavaScript step or even the Complex Object Editor (COE).

Glossary

assertion

An *assertion* is an element that runs after a step and all its filters have run. An assertion verifies that the results from running the step match the expectations. An assertion is typically used to alter the flow of a test case or virtual service model. Global assertions apply to each step in a test case or virtual service model. For more information, see Assertions in the *User Guide.*

asset

An *asset* is a set of configuration properties that are grouped into a logical unit. For more information, see Assets in the *User Guide*.

audit document

An *audit document* lets you set success criteria for a test, or for a set of tests in a suite. For more information, see Building Audit Documents in the *User Guide.*

companion

A *companion* is an element that runs before and after every test case execution. Companions can be understood as filters applicable to the entire test case as opposed to individual test steps. Companions are used to configure global (to the test case) behavior in the test case. For more information, see Companions in the *User Guide.*

configuration

A *configuration* is a named collection of properties that usually specify environment-specific values for the system under test. Removing hard-coded environment data enables you to run a test case or virtual service model against different environments simply by changing configurations. The default configuration in a project is named project.config. A project can have many configurations, but only one configuration is active at a given time. For more information, see Configurations in the *User Guide.*

Continuous Service Validation (CVS) Dashboard

The *Continuous Validation Service (CVS) Dashboard* lets you schedule test cases and test suites to run regularly, over an extended time period. For more information, see Continuous Validation Service (CVS) in the *User Guide.*

conversation tree

A *conversation tree* is a set of linked nodes that represent conversation paths for the stateful transactions in a virtual service image. Each node is labeled with an operation name, such as withdrawMoney. An example of a conversation path for a banking system is getNewToken, getAccount, withdrawMoney, deleteToken. For more information, see Conversation Editor Tree View in the *Virtual Services Environment Guide.*

coordinator

A *coordinator* receives the test run information in the form of documents, and coordinates the tests that are run on one or more simulator servers. For more information, see Coordinator Server in the *User Guide.*

data protocol

A *data protocol* is also known as a data handler. In LISA Virtual Services Environment, it is responsible for handling the parsing of requests. Some transport protocols allow (or require) a data protocol to which the job of creating requests is delegated. As a result, the protocol has to know the request payload. For more information, see Using Data Protocols in the *Virtual Services Environment Guide.*

data set

A *data set* is a collection of values that can be used to set properties in a test case or virtual service model at runtime. Data sets provide a mechanism to introduce external test data into a test case or virtual service model. Data sets can be created internal to LISA, or externally (for example, in a file or a database table). For more information, see Data Sets in the *User Guide.*

desensitize

Desensitizing is used to convert sensitive data to user-defined substitutes. Credit card numbers and Social Security numbers are examples of sensitive data. For more information, see Desensitizing Data in the *Virtual Services Environment Guide.*

event

An *event* is a message about an action that has occurred. You can configure events at the test case or virtual service model level. For more information, see Understanding Events in the *User Guide.*

filter

A *filter* is an element that runs before and after a step. A filter gives you the opportunity to process the data in the result, or store values in properties. Global filters apply to each step in a test case or virtual service model. For more information, see Filters in the *User Guide.*

group

A *group*, or a *virtual service group,* is a collection of virtual services that have been tagged with the same group tag so they can be monitored together in the VSE Console.

Interactive Test Run (ITR)

The *Interactive Test Run (ITR)* utility lets you execute a test case or virtual service model step by step. You can modify the test case or virtual service model at runtime and rerun to verify the results. For more information, see Using the Interactive Test Run (ITR) Utility in the *User Guide.*

lab

A *lab* is a logical container for one or more lab members. For more information, see Labs and Lab Members in the *User Guide.*

magic date

During a recording, a date parser scans requests and responses. A value matching a wide definition of date formats is translated into a *magic date*. Magic dates are used to verify that the virtual service model provides meaningful date values in responses. An example of a magic date is {{=doDateDeltaFromCurrent("yyyy-MM-dd","10");/*2012-08-14*/}. For more information, see Magic Strings and Dates in the *Virtual Services Environment Guide.*

magic string

A *magic string* is a string that is generated during the creation of a service image. A magic string is used to verify that the virtual service model provides meaningful string values in the responses. An example of a magic string is {{=request_fname;/chris/}}. For more information, see Magic Strings and Dates in the *Virtual Services Environment Guide.*

match tolerance

Match tolerance is a setting that controls how LISA Virtual Services Environment compares an incoming request with the requests in a service image. The options are EXACT, SIGNATURE, and OPERATION. For more information, see Match Tolerance in the *User Guide.*

metrics

Metrics let you apply quantitative methods and measurements to the performance and functional aspects of your tests, and the system under test. For more information, see Generating Metrics in the *User Guide.*

Model Archive (MAR)

A *Model Archive (MAR)* is the main deployment artifact in LISA. MAR files contain a primary asset, all secondary files that are required to run the primary asset, an info file, and an audit file. For more information, see Working with Model Archives (MARs) in the *User Guide.*

Model Archive (MAR) Info

A *Model Archive (MAR) Info* file is a file that contains information that is required to create a MAR. For more information, see Working with Model Archives (MARs) in the *User Guide.*

navigation tolerance

Navigation tolerance is a setting that controls how LISA Virtual Services Environment searches a conversation tree for the next transaction. The options are CLOSE, WIDE, and LOOSE. For more information, see Navigation Tolerance in the *Virtual Services Environment Guide.*

network graph

The network graph is an area of the Server Console that displays a graphical representation of the DevTest Cloud Manager and the associated labs. For more information, see Start a Lab in the *User Guide*.

node

Internal to LISA, a test step can also be referred to as a *node*, explaining why some events have node in the EventID.

path

A *path* contains information about a transaction that the LISA Java Agent captured. For more information, see Pathfinder Console - Paths Tab in the *Pathfinder Guide.*

path graph

The *path graph* is an area of the LISA Pathfinder Console that contains a graphical representation of a path and its components. For more information, see Path Graph in the *Pathfinder Guide.*

project

A *project* is a collection of related LISA files. The files can include test cases, suites, virtual service models, service images, configurations, audit documents, staging documents, data sets, monitors, and MAR info files. For more information, see Project Panel in the *User Guide.*

property

A *property* is a key/value pair that can be used as a runtime variable. Properties can store many different types of data. Some common properties include LISA_HOME, LISA_PROJ_ROOT, and LISA_PROJ_NAME. A configuration is a named collection of properties. For more information, see Properties in the *User Guide.*

quick test

The *quick test* feature lets you run a test case with minimal setup. For more information, see Stage a Quick Test in the *User Guide.*

registry

The *registry* provides a central location for the registration of all LISA Server and LISA Workstation components. For more information, see Registry in the *User Guide.*

service image (SI)

A *service image* is a normalized version of transactions that have been recorded in LISA Virtual Services Environment. Each transaction can be stateful (conversational) or stateless. One way to create a service image is by using the Virtual Service Image Recorder. Service images are stored in a project. A service image is also referred to as a *virtual service image* (VSI). For more information, see Service Images in the *Virtual Services Environment Guide.*

simulator

A *simulator* runs the tests under the supervision of the coordinator server. For more information, see Simulator Server in the *User Guide.*

staging document

A *staging document* contains information about how to run a test case. For more information, see Building Staging Documents in the *User Guide.*

subprocess

A *subprocess* is a test case that another test case calls. For more information, see Building Subprocesses in the *User Guide.*

test case

A *test case* is a specification of how to test a business component in the system under test. Each test case contains one or more test steps. For more information, see Building Test Cases in the *User Guide.*

test step

A *test step* is an element in the test case workflow that represents a single test action to be performed. Examples of test steps include Web Services, Java Beans, JDBC, and JMS Messaging. A test step can have LISA elements, such as filters, assertions, and data sets, attached to it. For more information, see Building Test Steps in the *User Guide.*

test suite

A *test suite* is a group of test cases, other test suites, or both that are scheduled to execute one after other. A suite document specifies the contents of the suite, the reports to generate, and the metrics to collect. For more information, see Building Test Suites in the *User Guide.*

think time

Think time is how long a test case waits before executing a test step. For more information, see Add a Test Step (example) and Staging Document Editor - Base Tab in the *User Guide.*

transaction frame

A *transaction frame* encapsulates data about a method call that the LISA Java Agent intercepted. For more information, see Transactions and Transaction Frames in the *Pathfinder Guide.*

virtual service model (VSM)

A *virtual service model* receives service requests and responds to them in the absence of the actual service provider. For more information, see Virtual Service Model (VSM) in the *Virtual Services Environment Guide.*

Virtual Services Environment (VSE)

The *Virtual Services Environment (VSE)* is a LISA Server application that you use to deploy and run virtual service models. For more information, see the *Virtual Services Environment Guide.*