Ingenieurbüro David Fischer AG | A Company of the Apica Group
http://www.apicasystem.com

ZebraTester

"Load Test Plug-In" Developer Handbook

Table of Contents

1 Overview

1.1 Document Contents

This Handbook consists of two parts.

Part One provides an overview of the Properties, Runtime Behavior, and Configuration of ZebraTester Load Test Plug-Ins.

Part Two provides information on how to develop ZebraTester "Load Test Plug-Ins".

1.2 Introduction

ZebraTester Load Test Plug-Ins are Extension Modules to the ZebraTester product. Load Test Plug-Ins are configured using the GUI, and are executed during a Load Test.

In addition to the pre-defined Load Test Plug-Ins delivered with the ZebraTester product, additional user-defined Plug-Ins can be developed to provide extra functionality to meet specific needs.

Plug-Ins have the major characteristic of being re-usable. Once developed, a Plug-In can be re-used in every Load Test program.

The basic framework of a Load Test Plug-In can be easily created by using the Wizard; however, the completion of the finished Plug-in - coding the necessary functionality - requires the ability to program in Java. Then, once the Plug-In has been developed, configuring it for use with a Load Test program does not require programming knowledge; Plug-In configuration is done with a few mouse-clicks in the GUI.

The execution of a Plug-In takes place on the same machine which is running the Exec-Agent, and the Load Test program itself. When a Load Test is started, the relevant Plug-In is automatically transmitted to the Exec-Agent with the Load Test program - no additional prior configuration of the Plug-In is necessary.

2 Properties, Runtime Behavior and Configuration

All Load Test Plug-Ins are bound to the Load Test program via the GUI, using the Variable Handler. Load Test Plug-Ins can:

1. Import GUI Variables
2. Execute its own program code
3. Export GUI Variables
4. Abort an executing Load Test
5. Insert additional measurement data into the Load Test Results.

Example 1: A Plug-in computes a date (MM.DD.YYYY) which is always three days in the future, relative to the current date. This date is exported as a GUI Variable, and the Variable is then used in an HTML Form as an input parameter for entering a dynamically-assigned "booking date".

Example 2: In an online ordering process, the "order number" is first extracted from a URL call as a GUI Variable, and then passed as input to a Plug-In. The Plug-In writes all "order numbers" of the simulated users into a file, which is used after the completion of the Load Test to cancel the simulated orders.

Example 3: A Plug-In continually monitors the progress of a Load Test, and aborts the Load Test if more than 90% of all Loops fail within a 5 minute period - computed over all virtual users. In this way, a long-running Load Test can be interrupted if a Web Server collapses and cannot itself recover.

2.1 Runtime Environment

Plug-Ins are tightly integrated with a Load Test, and provide access to:

- all GUI Variables defined in the Variable Handler. Importing and Exporting GUI Variables is supported.
- all real-time measurement data of a running Load Test program.
- the data from specific, or all, executed URL Calls of a running Load Test program. This includes the ability to modify and/or enhance the data before execution, as well as the ability to perform additional processing on the result data after execution.
- the operating system of the Exec-Agent machine, provided the interfaces are available via Java. This includes, for example, network connections and files on the file system or disk.
- additional data from the Runtime Environment of a Load Test, such as the current number of virtual users, the number of current Cluster Members running a Load Test (via Cluster Jobs), and the cookie storage of individual virtual users.

There are some restrictions; for example, a Plug-In cannot, in general, influence the execution path of a Load Test program. This means that a Plug-In cannot modify the order of the executed URL Calls. A Plug-In also cannot correct, after the fact, an error arising as the result of a (measured) URL Call. However, a Plug-In can turn a successful completed URL Call into a failed URL Call.

2.2 Plug-In Lifecycle during Test Execution

A Plug-In is initialized, executed one or more times, and then de-initialized.

Often the Initialization and De-Initialization of a Plug-In involves no special activities; that is, these steps are often "empty". On the other hand, during a Load Test a Plug-In could be programmed to perform various functions; for example, accessing a database. In such cases, the connection to the database would be opened during the one-time Plug-In Initialization, used during the Load Test to read or write data, and then closed during the one-time Plug-In De-Initialization.

The timing of the Plug-In Initialization, De-Initialization, and Execution can all be **configured separately** and can **occur at different times**. This timing is tightly coupled with the data the Plug-In has access to, and is referred to as the **Plug-In Scope**.

The following **Plug-In Scopes** are available:

- **global**: immediately before, or immediately after, the execution of a Load Test program
- **user**: when a virtual user is created, or when a virtual user has ended
- **loop**: at the beginning, or at the end, of a Loop
- **URL**: before, or after, a Call to a specific URL, as well as before/after all URL Calls.

The most commonly-used Scope during Plug-In Initialization and De-Initialization is **global**. The most commonly-used Scope during the execution of a Plug-In is either **loop** or **URL**.

List of possible Plug-In Initialization Scopes:

Initialization Scope	Plug-In Initialization	Plug-In De-Initialization
global	Immediately after the start of a Load Test program	Immediately before the end of a Load Test program
user	Before the creation of each virtual user, and before the execution of the user's first Loop	After the end of each virtual user, and after the execution of the user's last Loop
loop	Before the execution of each Loop	After the execution of each Loop. Note: De-Initialization also occurs if the Loop fails.
URL	This Scope cannot be used during the Initialization or De-Initialization phases of the Plug-In lifecycle.	

List of possible Plug-In Execution Scopes:

Execution Scope	Before or After	Plug-In Execution
global	**before**	Immediately at the start of a Load Test program, but after Plug-In Initialization
global	**after**	Immediately before the end of a Load Test program, but before Plug-In De-Initialization
user	**before**	Before the creation of each virtual user, before the user's first Loop execution
user	**after**	After the end of each virtual user, after the user's last Loop execution
loop	**before**	Before each Loop execution, once for each virtual user
loop	**after**	After each Loop execution, once for each virtual user[1]
URL	**before**	Before the execution of a specific URL Call, or before each URL Call
URL	**after**	After the execution of a specific URL Call, or after each URL Call[2]

[1] the Plug-In will not be executed if the relevant Loop fails; however, Plug-In De-Initialization is always performed.

[2] the Plug-In will always be executed, even if a URL Call results in an error.

Note: the Import of GUI Variables occurs immediately before Plug-In execution, and the Export of GUI Variables occurs immediately after Plug-In execution.

2.3 Plug-in Configuration using the GUI

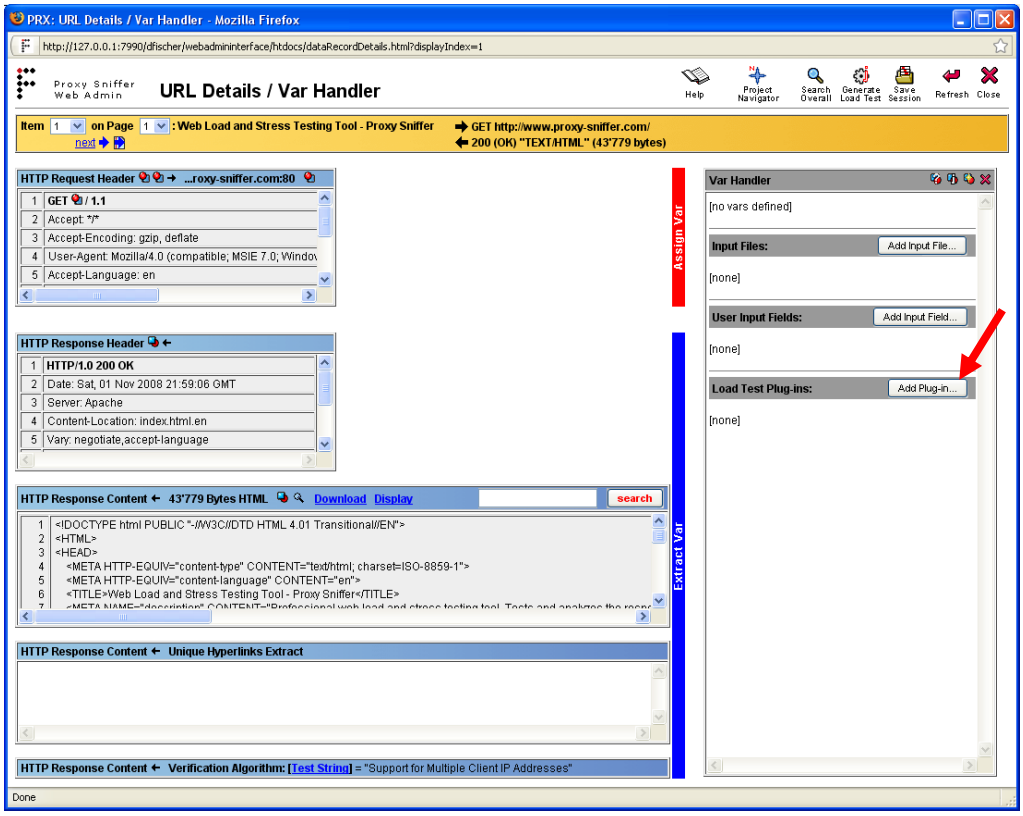
All available Plug-Ins can be found under **MyTests** in the **Plugins** sub-directory.

During Plug-In configuration using the GUI (in the Variable Handler), ZebraTester looks in this directory, and provides a selectable list of available Plug-Ins. If a new Plug-In is received - by E-Mail for example - the Plug-In must first be copied into the **Plugins** sub-directory under **MyTests** before it can be selected in the GUI.

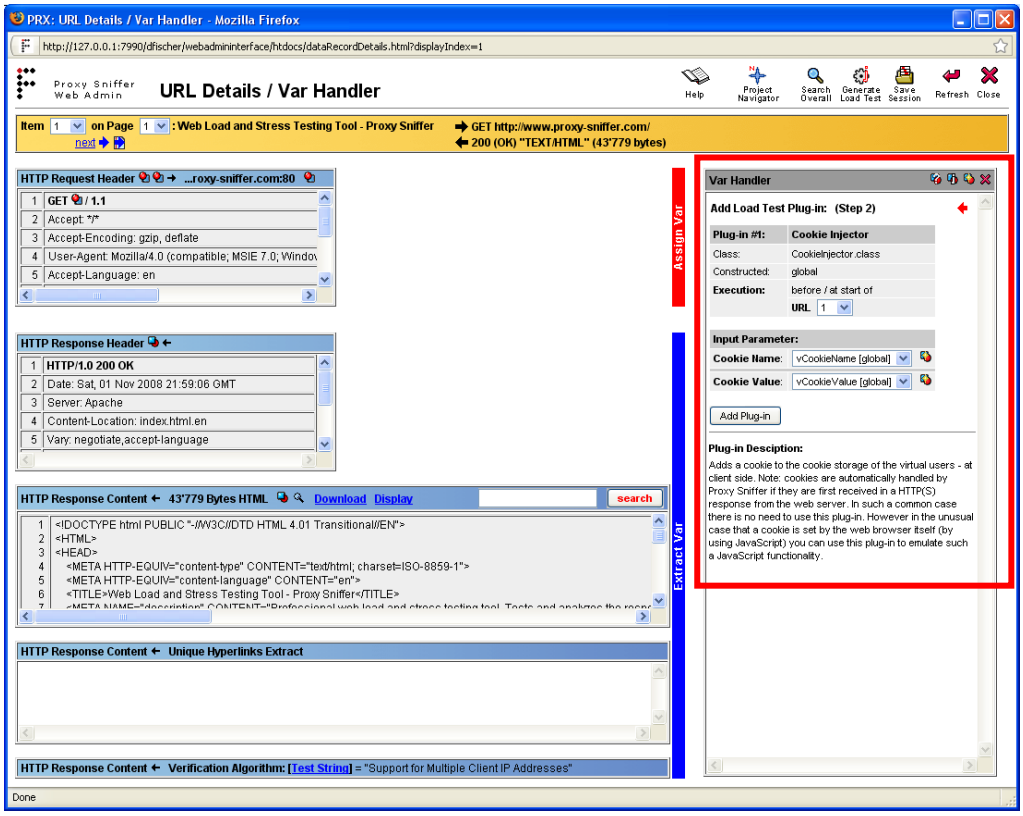
Plug-In configuration (that is, adding a Plug-In to a Load Test program) is done via the GUI's Variable Handler - similar to the definition of an Input File or "User Input Fields" - by using the "Add Plug-in..." button.

After the Plug-In has been selected in the GUI, the GUI Variables to be imported and/or exported by the Plug-In must be defined, and the Initialization and Execution Scope must be specified. Most Plug-Ins will pre-define, or hard-code, the Scopes during development; in this case, the Scopes will not be selectable in the GUI. An additional Plug-In property, which can be defined during development, indicates whether or not the Plug-In can be used more than once in the same Load Test program. If a Plug-In is defined to be usable only once per Load Test program, the GUI will prevent it from being configured more than once for use with a given Load Test program.

Picture: Adding a Plug-In to a Load Test Program:



Picture: Configuring the Imported GUI Variables of a Plug-In:



2.4 Plug-In Lifecycle after Configuration

When a Plug-In is configured using the GUI, the compiled Plug-In Java code will be saved in the same file that the Web Session uses for Web Session storage (in the "*.prxdat" file of the Web Session). If the Web Session "*.prxdat" file is stored on another computer, the Plug-In Java code is transferred there as well, and is then available on the other computer.

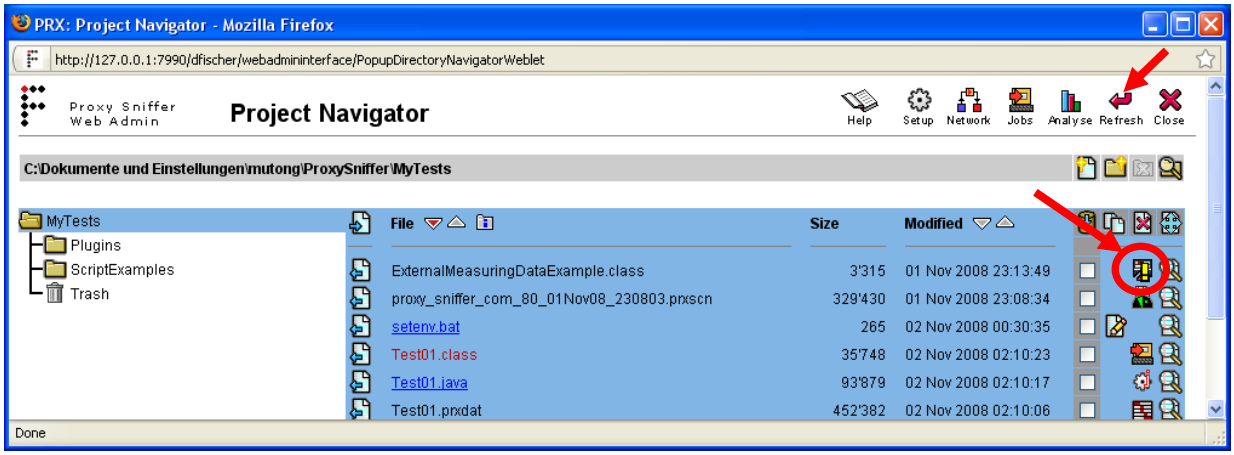
When a Load Test program is created, the Plug-In is automatically copied out of the Web Session, and a copy of the Plug-In is made in the Project Navigator directory in which the Load Test program was created. Before Test Execution, the compiled Load Test program ("*.class" file), including the Plug-In, is zipped. The resulting zip archive itself can be executed, and the Plug-In will be automatically transferred to the Exec Agent along with the Load Test program.

During Plug-In development, a new version of the Plug-In may be created in the **Plugins** sub-directory under **MyTests**. The ZebraTester GUI monitors this process, and will detect when a Web Session contains an older version of a Plug-In. When a new Plug-In is detected, the GUI will display a user dialog requesting that the new Plug-In be imported into the Web Session. Note that a new Plug-In version can only the imported when the number of GUI Input and Output Variables remains unchanged and retain the same meaning. If this is not the case, the Plug-In must first be manually removed using the Var Handler, and then re-imported to the Web Session.

In addition to the Web Sessions, the GUI monitors the compiled Plug-Ins in the Load Test program directories. After a "refresh" in the Project Navigator, older Plug-In versions will be marked with a yellow exclamation point. The updated Plug-In version can be copied from the **Plugins** sub-directory under **MyTests** to the Load Test program directory by clicking on the corresponding Plug-In's icon in the Project Navigator. If the number of GUI Input and Output Variables - including their meaning - in the new Plug-In version is the same as the previous version, the Load Test program can be started with the new Plug-In version, without the need for re-creating or re-compiling the Java code.

The internal comparison to see if a Plug-In is obsolete does not use the file date, but computes a checksum over the compiled Plug-In code. If the checksums are different, the "new" Plug-In is always considered to be that version currently in the **Plugins** sub-directory of **MyTests**.

Example: Old Plug-In version in the Load Test program Directory – Icon contains a yellow exclamation point



3 Developing your own Plug-In

The first step in developing a new Plug-In is to bind to the ZebraTester Runtime Environment. This is done using a Wizard which automatically creates a basic Plug-In program framework. This initial generated Plug-In code is able to be compiled on its own, but does not contain any inner logic to provide Plug-In functionality.

The second step is to develop the inner logic of the Plug-In. This must be done manually by coding in Java.

The third, and final step, is to add the Plug-In to a recorded Web Session by using the GUI Variable Handler.

3.1 Java API Documentation

The class libraries of the Java SDK 6.0 can be used for Plug-In coding. Additional, ZebraTester specific classes are in package **dfischer.utils** available.

The API documentation for the **dfischer.utils** package can be accessed on Windows systems via Start ➡ All Programs ➡ ZebraTester ➡ Documentation ➡ **ZebraTester API Javadoc**

On Mac OS X and Linux systems the API documentation is installed in the ZebraTester installation directory ➡ Documentation ➡ javadoc

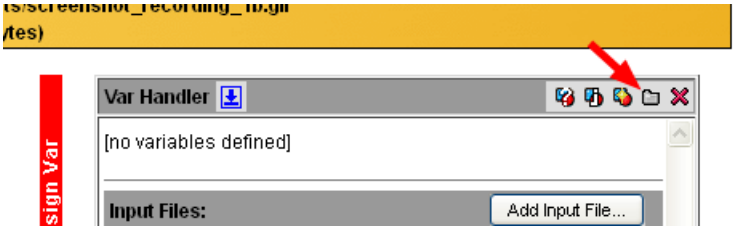
3.2 Using Multiple Classes and External Class Libraries

If a Plug-In consists of a single class and itself uses no external libraries, there is nothing further to be done, and this section can be skipped.

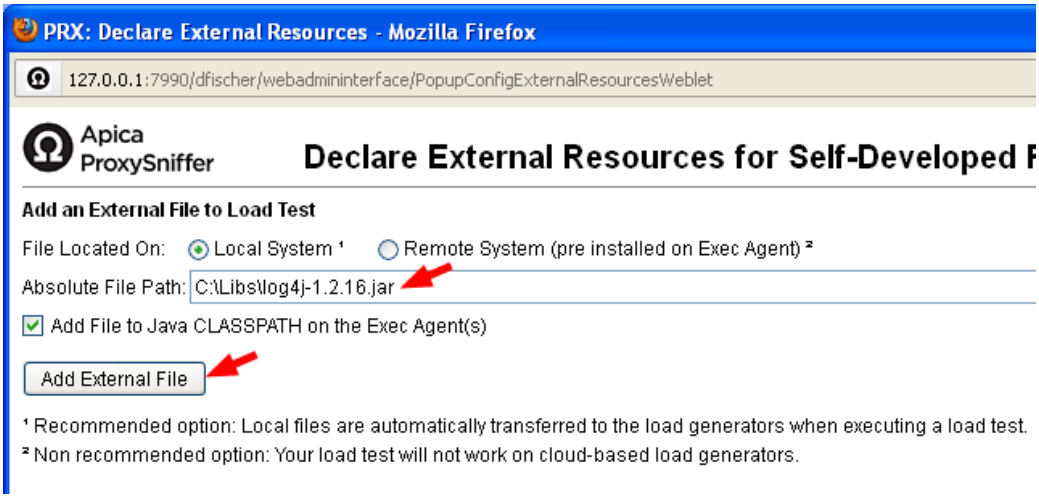
If a Plug-In requires the use of additional class libraries - in addition to those available in Java JDK 6.0 and package **dfischer.utils** (such as a database driver) - or if a Plug-In consists of more than one class, these must be available as "*.jar" files before beginning the development of the Plug-In Main class. Note that the Plug-In Main class can never be part of the Java package, and must not itself be contained in a JAR Archive.

If additional "*.jar" files are required, proceed as follows:

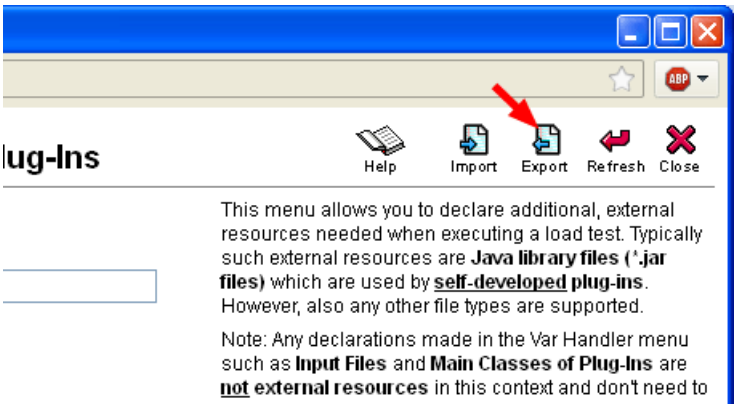
1. First load any recorded session.
2. Call the "Declare External Resources" Menu from the "Var Handler" menu:



3. Declare (add) all required jar files as External Resources:



4. Export the Declarations (they can later be imported in other recorded sessions):



5. You can now start developing your plug-in (see next chapter).

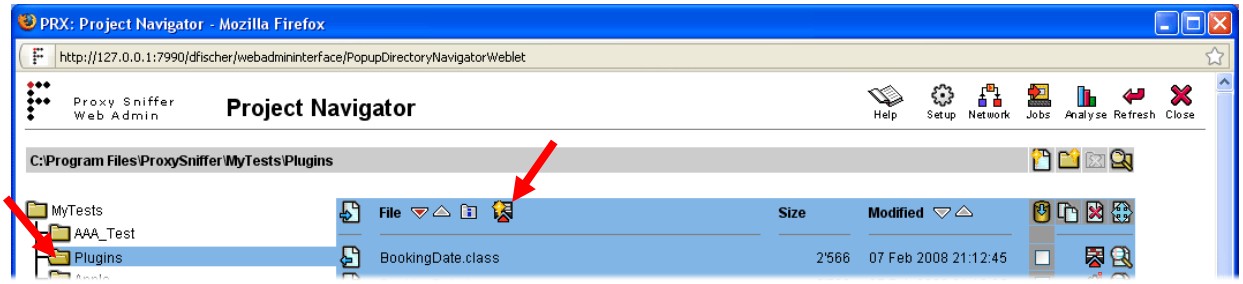
Note: Starting from ZebraTester version 5.0 it's no longer required to make any additions to the CLASSPATH of the Web Admin GUI. The revised Java "class loader" of ZebraTester will load all declared jar files instantly (on the fly).

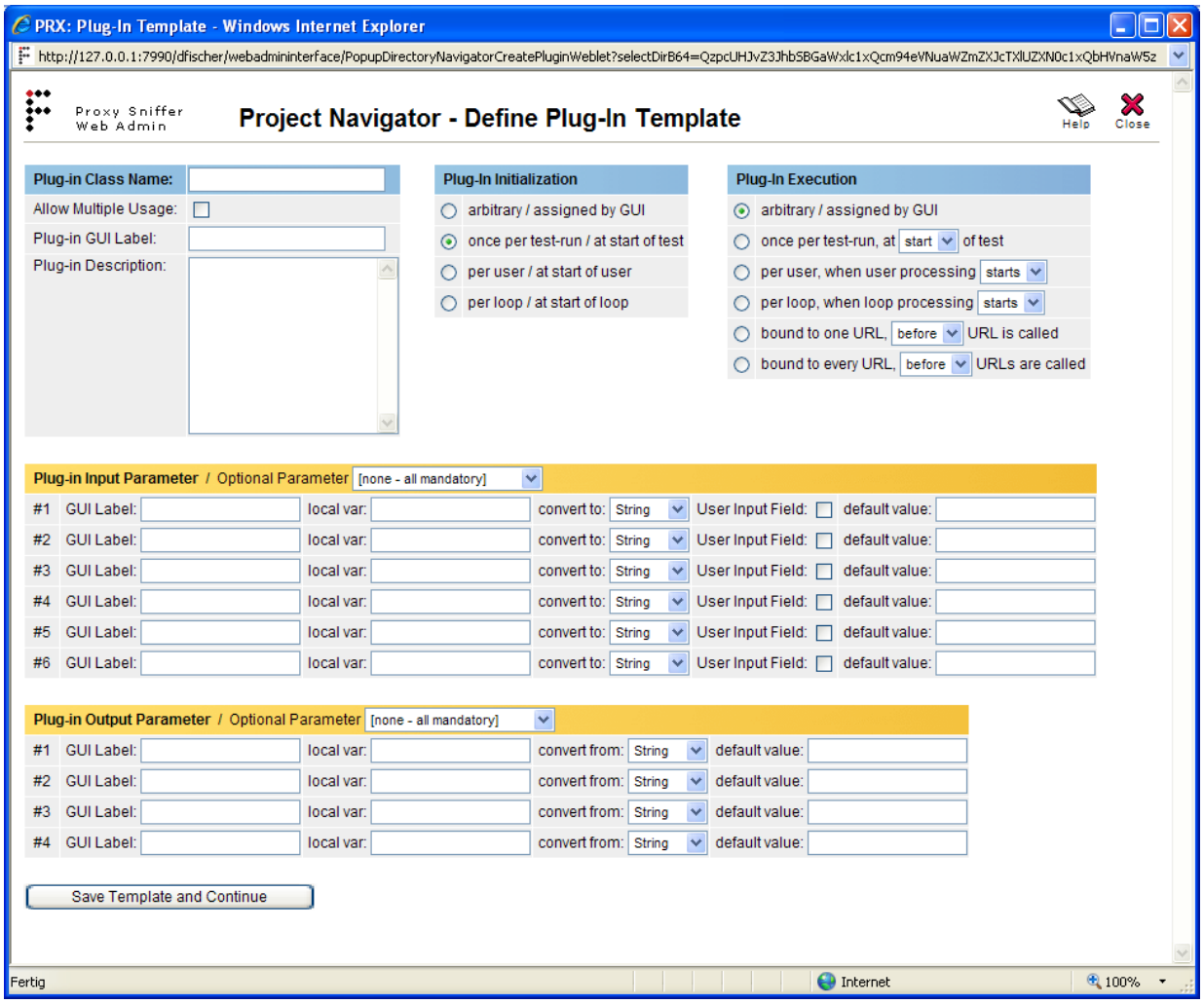
All declarations for external local resources are always stored together with a recorded session (*.prxdat file) - and also only visible for this recorded session. There is no system-wide or product-wide declaration available. Therefore, if another recoded session uses the same plug-in, you have to add the declarations for the external resources again to this other session. Alternatively, you can also export the declarations and import them later by using the import and export icons which are located at the top of the window.

Once a load test program is generated, all external resources are automatically zipped together with the load test program, and the whole ZIP-archive is automatically transferred to the load generators (Exec Agents). There is no need to pre-install the external resources on the load generators.

3.3 Creating the Program Skeleton using the Wizard

The Wizard is available in the Project Navigator **Plugins** sub-directory under **MyTests**, and can be started by clicking on its icon in the Project Navigator:





The following values must be entered in the box at upper left of the above form:

- **Plug-In Class Name:** name of the Plug-In Java class, without the "*.class" or "*.java" file extension.
- **Allow Multiple Usage**: indicates if the Plug-In can be added just once, or many times, in the same Load Test program or Web Session.
- **Plug-In GUI Label**: short text (one or two keywords) for the label of the Plug-In. This label will appear in the GUI associated with this Plug-In.

- **Plug-In Description**: detailed description of the Plug-In, also displayed in the GUI. Can consist of many sentences; however, HTML formatting is not supported. The only extra formatting supported here is "\n" for a line feed in the displayed text.

In the box positioned at the upper middle of the form, labelled "**Plug-In Initialization**", the Initialization Scope can be defined (see Section 2.2 of this document for a description of Plug-In Scopes). If "**arbitrary / assigned by GUI**" is selected, the Initialization Scope will not be pre-defined, and will be selectable in the GUI at the time the Plug-In is added. Note - it is recommended that this option NOT be selected; that is, whenever possible the Plug-In Initialization Scope should be pre-defined.

In the box positioned at the upper right of the form, labeled "**Plug-In Execution**", the Execution Scope can be defined (see Section 2.2 of this document for a description of Plug-In Scopes). The "**arbitrary / assigned by GUI**" option is also available for this Scope; and as with Initialization Scopes, a pre-defined Execution Scope is recommended.

Next, in the area further down in the middle of the form, the "**Plug-In Input Parameter**" definitions can be entered. A maximum of 6 Input Parameters can be defined. Input Parameters can be defined to be imported as "Mandatory GUI Variables" or "Optional GUI Variables" using the "**Optional Parameter**" drop-down list. Note that all definitions of "Mandatory GUI Variables" must precede any definitions of "Optional GUI Variables".

The form input fields for Plug-In Input Parameters are as follows

- **GUI Label:** short text (one or two keywords) for the description of the Parameter which will appear in the GUI.
- **local var**: name of the local Plug-In Variable in the automatically-generated Plug-In Java program code. At configuration time, the value of the GUI Variable is copied to this local Plug-In Variable.
- **convert to**: defines the data type of the local Plug-In Variable. Note that all GUI Variables in the Variable Handler are passed to the Plug-In as strings. "Convert To" = "int" indicates that the imported GUI Variable will be converted to the data type of the (local) "int" Variable.
- **User Input Field**: if selected, during Plug-In configuration in the GUI, a new "User Input Field" will be automatically created, and the value of the new GUI Variable will be set to the value of the local Plug-In Variable. If not selected, the Input Parameter GUI Variable must be manually selected in the GUI at Plug-in configuration time.
- **default value**: sets the default value of the local Variable in the Plug-In Java program code. If no default value is set, the local Variable will be initialized with the following default values:

Data Type	Initialized Value
String	“” (empy string)
int	-1 (minus one)
long	-1 (minus one)
float	-1 (minus one)
double	-1 (minus one)
boolean	false

Note: An imported boolean variable will have the value true if the string value of the corresponding GUI variable contains "1“ or "true“.

In the lower part of the form, the **Plug-In Output Parameter** definitions can be entered. A maximum of 4 Output Parameters can be defined. As for Input Parameters, Output Parameters can be mandatory or optional.

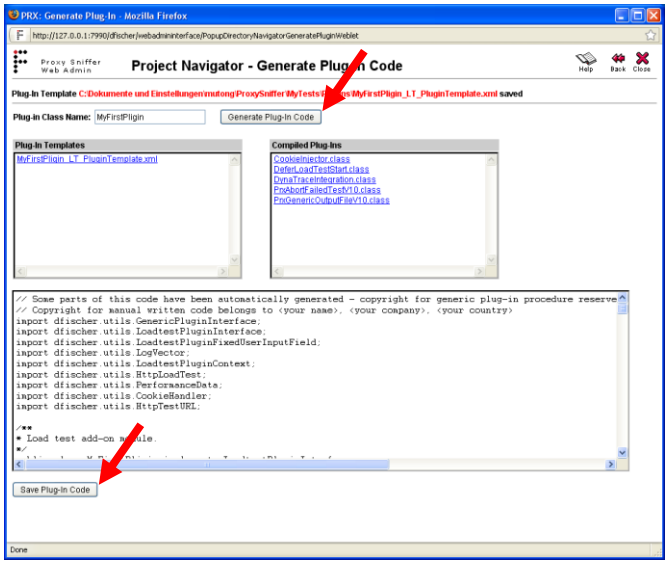
The form input fields for Plug-In Output Parameters are as follows:

- **GUI Label:** short text (one or two keywords) for the description of the Parameter which will appear in the GUI.
- **local var**: name of the local Plug-In Variable which will be later assigned the value of the corresponding configured GUI Variable.
- **convert from**: defines the data type of the local Variable in the Plug-In Java program code. Note that all local Variables are exported, after Plug-In execution, to the GUI Variables as strings.
- **default value**: sets the default value of the local Variable in the Plug-In Java program code. If no default value is set, the local Variable will be initialized in the same way as for Input Parameters (see above table).

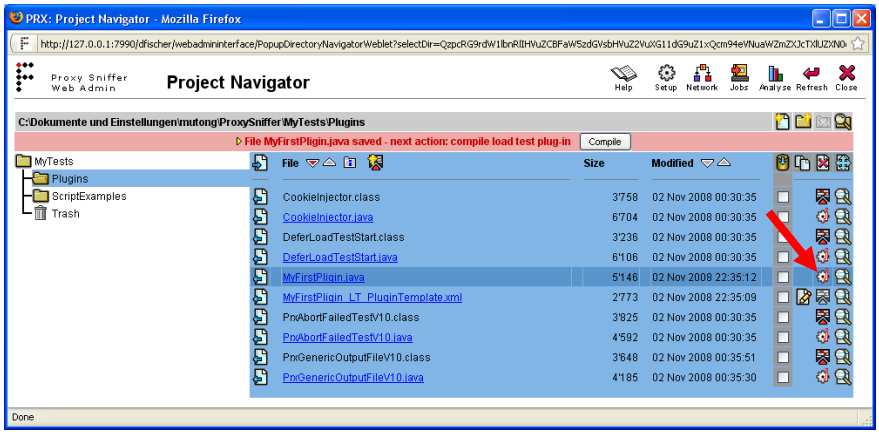
After all values have been defined, click the "**Save Template and Continue**" button at lower left in the form. In the resulting updated content of the same window, click the "**Generate Plug-In Code**" button.

The Plug-In program code is automatically generated in memory. To save the code to a file in the **Plugins Directory**, click on the "**Save Plug-In Code**" button at lower left in the window.

At this time, the Plug-In can be compiled for the first time in the Project Navigator. To re-compile the Plug-In at any time, click on the grey gear-wheel icon containing a "j" (at the right of the "*.java" Plug-In file name) in the Project Navigator.



Picture: Creating a Program Skeleton ("*.java" Code)



Picture: Re-compiling the Plug-In

3.4 Programming Plug-In Functionality

The Wizard is used (among other things) to create the following Plug-In methods:

- **Initialization** `public void construct(Object context)`

```
public void construct(Object context)
{
    // LoadtestPluginContext pluginContext = (LoadtestPluginContext) context;
}
```

- **Execution** `public void execute(Object context)`

```
public void execute(Object context)
{
    logVector = new LogVector();
    LoadtestPluginContext pluginContext = (LoadtestPluginContext) context;
}
```

- **De-Initialization** `public void deconstruct(Object context)`

```
public void deconstruct(Object context)
{
    // LoadtestPluginContext pluginContext = (LoadtestPluginContext) context;
}
```

Note: Usually only the **Execution** `execute` method will need to be enhanced with program code.

Depending on the Execution Scope, the Wizard will create sample code in the `execute` method. This sample code is enclosed by *//* `vvv --- sample code --- vvv` and

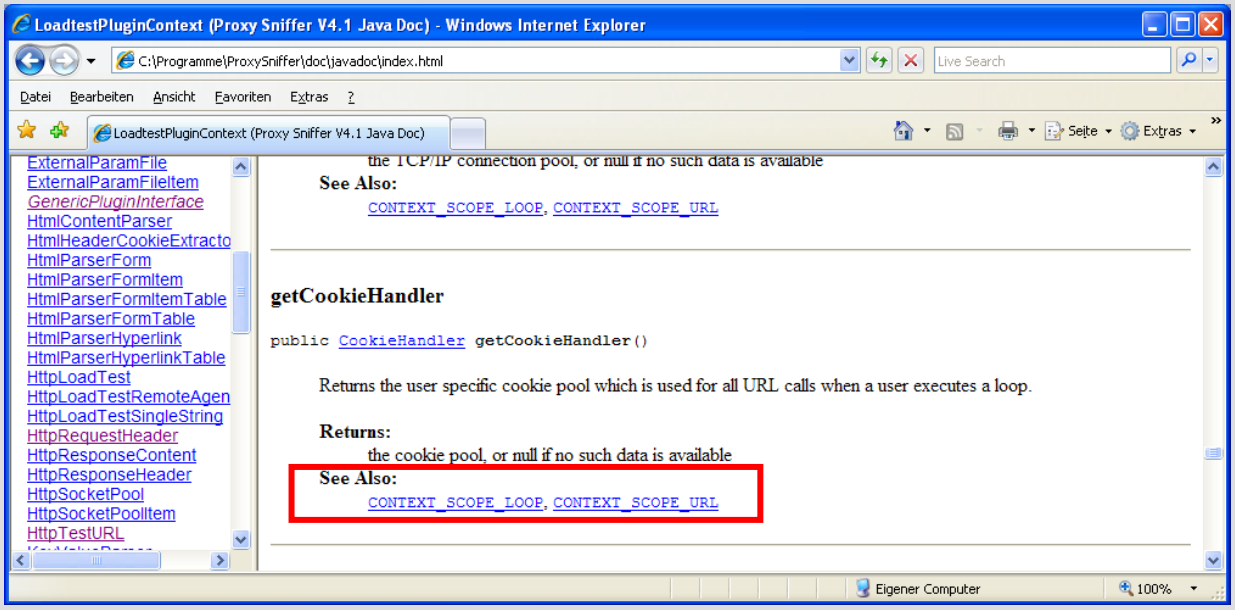
// `^^^ --- sample code --- ^^^`. This code must be replaced by own program code, and removed.

Access from the Plug-In to the Runtime Environment of a Load Test program occurs via the class **LoadtestPluginContext**, and the current instance of this class is passed to each Plug-In method.

A full description of the **LoadtestPluginContext** class can be found in the **ZebraTester Java API Documentation**.

Important: Methods within the **LoadtestPluginContext** class may not return a valid return value, depending on how the Plug-In Initialization Scope and Execution Scope are chosen (`CONTEXT_SCOPE_GLOBAL`, `CONTEXT_SCOPE_USER`, `CONTEXT_SCOPE_LOOP` or `CONTEXT_SCOPE_URL`). The **ZebraTester Java API Documentation** provides a description for each method, under "**See Also**", which specifies the Scopes for which the method returns valid values during Plug-In execution.

Example: the `getCookieHandler()` method only returns a valid Return Value when the Scope is `CONTEXT_SCOPE_LOOP` or `CONTEXT_SCOPE_URL`.



More information on Scopes can be found in section 2.2 of this document.

If the Initialization Scope or Execution Scope is not set using the Wizard (**arbitrary / assigned by GUI** option selected), these must be specified during Plug-In execution. The method `getContextScope()` has been provided for this purpose.

3.4.1 The "LoadtestPluginContext" Class

This section contains a description of a few selected methods of the **LoadtestPluginContext** class. Full documentation for this class can be found in the **ZebraTester Java API Documentation**.

Method	Description	Allowed Scopes
`getContextScope()`	Returns the current Scope at Runtime. Only necessary when the Scope is NOT set using the Wizard.	all Scopes
`getPerformanceData()`	Provides access to the current measurement data. Example: The method `PerformanceData.getPassedLoops()` can be used to determine how many Loops, measured over all virtual users, have been successfully executed up to the current time.	all Scopes
`getHttpLoadTest()`	Provides access to additional data from the Runtime Environment. For details, see the next section.	all Scopes
`getHttpTestURL()`	Provides access to data from the currently running URL Call. Example: Using the method `HttpTestURL.getRequestHeaderObject().appendHeader Field()` an additional HTTP Request Header Field can be inserted into a URL Call.	`CONTEXT_SCOPE_URL`
`getThreadStep()`	Returns the index of the currently executing URL Call. Example 1: the index can be used as an Input Parameter to the method `PerformanceData.getPerformanceDataRecord()` in order to access detailed statistical measurement data from the URL. Example 2: the index can be used as an Input Parameter to the method `PerformanceData.getPageInfoTextOfUrl()` in order to retrieve the text of a Page Break belonging to a URL.	`CONTEXT_SCOPE_URL`
`setContinueInnerLoopFlag()`	Forces a continue in a Inner Loop (jump back at the start of the Inner Loop)	`CONTEXT_SCOPE_URL`
`setBreakInnerLoopFlag()`	Forces a break in a Inner Loop (jump out of the Inner Loop)	`CONTEXT_SCOPE_URL`
`markUrlAsFailed(<Text>)`	Turns a successful passed URL Call into a failed URL Call (trigger a red colored error)	`CONTEXT_SCOPE_URL`

3.4.2 The "HttpLoadTest" Class

This section contains a description of a few selected methods of the **HttpLoadTest** class. Full documentation for this class can be found in the **ZebraTester Java API Documentation**.

Method	Description
`contentToDiskFile()`	Writes the content of a executed URL Call (HTTP Response content) to a file on disk.
`getClusterMemberLocalId()`	For Cluster Jobs, this method returns the number of the relevant Cluster Member (Exec Agent). Cluster Member numbering is zero-based; therefore, the first Cluster Member will have the number = 0, the second will have the number = 1, and so forth. With a Cluster Job, the same Plug-In is executing in parallel on each Cluster Member; therefore, it may be necessary (with the help of this method) to restrict certain Plug-In actions to a specific Cluster Member.
`triggerAbort()`	Causes an abort of a running Load Test program, without losing the statistical measurement data collected to the point of interruption; that is, the "*.prxdat" file will still be created.

3.4.3 Additional Details about the Runtime Environment

3.4.3.1 Debug Output during Plug-In Execution

The standard **System.out.println()** method should never be used for debug output inside the Plug-In **execute** method, as this would lead to log messages for different virtual user Loops being jumbled together in the Load Test Job log file. The **logVector.log()** method should be used instead.

Note: In the **construct** and **deconstruct** methods it is not possible to use the **logVector.log()** method; therefore, only the **System.out.println()** method can be used in these methods.

In order to have a complete debug output from a Load Test program, including debug output from any Plug-Ins, the field "**Debug Options**" must be set to "**debug loops (including var handler)**" in the GUI menu "Project Navigator - Execute Load Test" at the start of a Load Test.

Note: For a later execution of an actual Load Test, this option should be reset to "**none -recommended**".

Debug output is written to the corresponding Load Test Job "*.out" file. If a program crash occurs during a Load Test due to an error in a Plug-In, the output is written to the Load Test Job "*.err" file.

3.4.3.2 Initializing a Plug-In using Imported GUI Variables

Imported GUI Variables of a Plug-In are transferred (each time) immediately before execution; that is, immediately before the Plug-In **execute** method is called.

A situation can be imagined where a Plug-In would need access to a GUI Variable during Plug-In Initialization; that is, when the **construct** method is called. This is not supported. A work-around would be to perform a "lazy" Initialization, where the **construct** method does not actually do any Initialization, and the **execute** method first checks to see if the Plug-In has been Initialized. In the first call to the **execute** method this would not be true, and the Plug-In could be Initialized using a GUI Variable. Each subsequent call to the **execute** method would skip the Initialization. An example code fragment is given below:

```
[...]

/**
* Load test add-on module.
*/
public class TestPlugin implements LoadtestPluginInterface
{
    private String vInitialParameter = "";  // input parameter #1
    private int vNormalInputParameter = -1; // input parameter #2
    private boolean pluginInitialized = false;

    private String vOutputParam = "";       // output parameter #1

    private LogVector logVector = null;     // internal log vector

[...]

    public void construct(Object context)
    {
    }

[...]

    public void execute(Object context)
    {
        logVector = new LogVector();
        LoadtestPluginContext pluginContext = (LoadtestPluginContext) context;
```



```
        if (!pluginInitialized)
        {
            // initialize plug-in by using parameter #1
            doInitialize(vInitialParameter);

            pluginInitialized = true;
        }

        [...]

    }

[...]
```

3.4.3.3 Extracting HTTP Response Data from URL Calls

If it is necessary to extract data from the HTTP Response Header or HTTP Response Content of a URL Call, the Plug-In Execution Scope **URL / after** can be specified. In this case, care must be taken when programming the **execute** method because the Plug-In will also be called when a URL Call fails; that is, there may be no data for the Plug-In to process. Consider the example when a timeout occurs and there is no Response Data, or if instead of a 200 HTTP Response Code, a 500 "Internal Server Error" is received from the Web Server.

In order to avoid a Plug-In "crash", which would bring down the Load Test program as well, a check should be made to ensure that a valid Server Response has been received before trying to extract data from the HTTP Response.

To assist in this process, the **LoadtestPluginContext** class has a method **urlPassed().** This method returns **true** if and only if during a Load Test the current URL Call does not return an error, as defined in the GUI Configuration Menu "HTTP Response Verification".

In this case, "does not return an error" means:

- The correct HTTP Response Code was received by the URL Call
- The Response contained the correct MIME type (e.g. TEXT/HTML)
- The verification of the received data content, in terms of data size or the presence of a specified text fragment, was successful

The following code fragment illustrates the correct way to access the HTTP Response data content of a URL Call:

```
public void execute(Object context)
{
    LoadtestPluginContext pluginContext = (LoadtestPluginContext) context;

    if (pluginContext.urlPassed())
    {
        HttpTestURL httpTestURL = pluginContext.getHttpTestURL();
        String content = httpTestURL.getContentString();

        [...] // Processing of the Response Data
    }
    [...]
}
```


3.4.3.4 Plug-In Execution at the End of a Loop

Note that a Plug-In is not executed at the **end of a Loop** if an exception occurs during a URL Call inside the Loop, as this would cause the entire Loop to fail. This means that if an error occurs, and the Plug-In Execution Scope is **loop / after**, the Plug-In **execute** method will not be executed.

If the Plug-In also has the Initialization Scope **loop**, the **deconstruct** method will always be called. This will occur for this Initialization Scope at the end of each Loop, whether or not an error occurred in the Loop.

In this case, the Plug-In **deconstruct** method will always receive a current instance of the **LoadtestPluginContext** class. Using the **loopPassed()** method, it can be determined, from inside the **deconstruct** method, if a Loop failed or not. In the case of a failed Loop, access is be provided to the URL Call which caused the failure by calling method **getHttpTestURL()**.

3.4.3.5 Using a Plug-In in Cluster Jobs

In the case of Cluster Jobs, a Plug-In will execute in parallel in all Exec Agents of the Cluster.

If a special action is required to execute only once for the entire Cluster Job, the Plug-In must be bound to a single Exec Agent in the Cluster. To do so, the Plug-In can access the required Exec Agent ID Number by calling the method **HttpLoadTest. getClusterMemberLocalId()**. This method returns zero-based Exec Agent IDs; that is, the first Cluster Member is Number 0, the second is Number 1, and so on.

As a Cluster can theoretically consist of a single Exec Agent, the Plug-In should be bound to the first Cluster Member (ID = 0) to ensure it is always executed. In the case of normal Exec Agent Jobs, which are not executed using a Cluster, the method **getClusterMemberLocalId()** returns the value -1.

The following code fragment demonstrates how a single action per Job can be executed from a single Exec Agent, irregardless of whether the Job is an Exec Agent Job or a Cluster Job:

```
public void execute(Object context)
{
    LoadtestPluginContext pluginContext = (LoadtestPluginContext) context;
    HttpLoadTest httpLoadTest = pluginContext.getHttpLoadTest();

    int clusterMemberLocalId = httpLoadTest.getClusterMemberLocalId();
    if ((clusterMemberLocalId == -1) || (clusterMemberLocalId == 0))
    {
        // action is executed by only one Exec Agent
        [...]
    }
}
```


3.4.3.6 Integrating Additional (External) Measurement Data

Using a Plug-In, it is possible to collect additional measurement data during a Load Test, and display these data later in the GUI (Menu "Load Test Result Detail - Statistics and Diagrams").

In this case, it does not matter if the measurement data stems from internal data sources (such as data extracted from a URL Response in the Load Test program) or from external data sources. The only condition is that these additional measurement data be available during the execution of the Load Test program. Measurement data from external systems can also be included in the Load Test Results ("*.prxres" file), provided the appropriate Plug-In programming is performed.

The last point in time that additional measurement data can be integrated is just before the end of the Load Test program; that is, when each virtual user has completed its last Loop. The relevant Plug-In Initialization Scope is therefore **global**, and the insertion of the additional measurement data occurs in the Plug-In **deconstruct** method.

The collection of additional measurement data can be done at any time during the Load Test, provided that for each measurement an instance of the class **dfischer.utils.DataCollectionSequence** is first created. The caption in the GUI for the related Measurement Diagram is provided to the constructor of this class. The parameters to this constructor are as follows:

- sequenceId: an arbitrary, unique number in the data sequence
- diagramTitle: title of the Diagram in the GUI
- diagramSubTitle: subtitle of the Diagram in the GUI
- yAxisLabel: caption of the Y-Axis in the Diagram
- sequenceContext: not used in this case, always set to null

After an instance of **DataCollectionSequence** has been created, the individual measurement values of the Measurement can be inserted using repeated calls to the method **addItem(DataCollectionFloatItem** floatItem**)**. An instance of the class **DataCollectionFloatItem** must be created before each call to **addItem**, and the constructor for **DataCollectionFloatItem** has the following two parameters:

- timeStamp: time of the measurement, expressed in milliseconds since 1970. If the current time is desired, **System.currentTimeMillis()** can be used.
- floatValue: Measurement value.

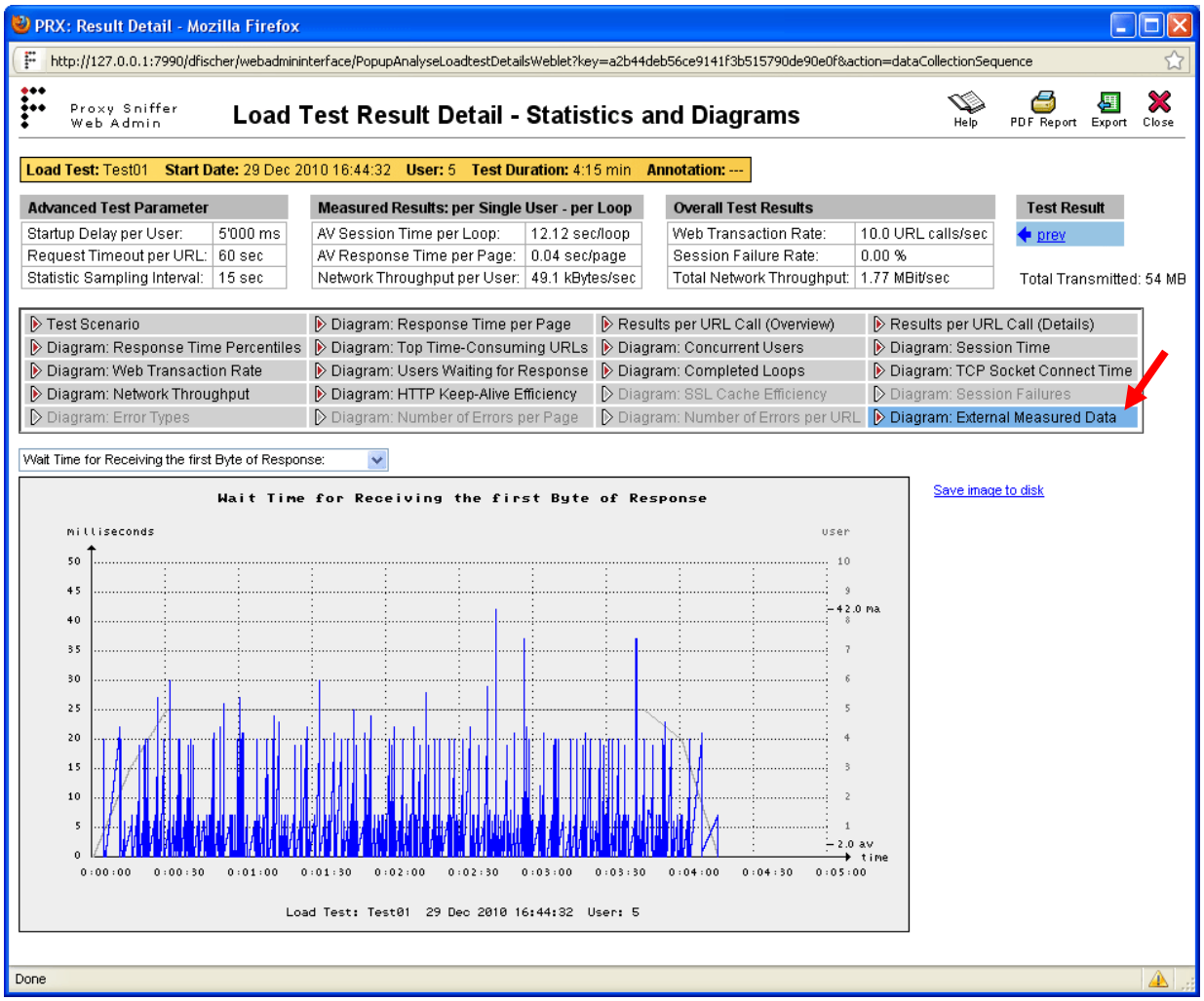
Note: Ensure that measurements occurring before the start of the Load Test program are not included in the GUI Diagram. If data, including timestamps, from external systems are imported, the Plug-In should first determine the time difference between the local system and the external system, and use this difference to adjust the external system timestamp.

After all data has been collected, these data must be inserted into the Load Test Result. This is done by repeatedly calling the method **PerformanceData.addDataCollectionSequence(DataCollectionSequence)**, once per Measurement; that is, once per instance of **DataCollectionSequence**. Access to the current instance of **PerformanceData** is provided by the Plug-In Context, which is an instance of the class **LoadtestPluginContext** provided as a parameter to the Plug-In **construct**, **execute**, and **deconstruct** methods.

Example 1: The following simple example shows how to collect additional measurement data from an URL Call and how to add these data to the load test result:

```
[...]
import dfischer.utils.DataCollectionSequence;
import dfischer.utils.DataCollectionFloatItem;


/**
 * Load test add-on module.
 */
public class AdditionalDataExample implements LoadtestPluginInterface
{
    private LogVector logVector = null;

    private DataCollectionSequence simpleSequence = new DataCollectionSequence(1,
"Wait Time for Receiving the first Byte of Response", "", "Milliseconds", null);


    [...]


    /**
     * Execute plug-in after URL call.
     *
     * Intrinsic plug-in implementation.
     */
    public void execute(Object context)
    {
        logVector = new LogVector();
        LoadtestPluginContext pluginContext = (LoadtestPluginContext) context;

        HttpTestURL httpTestURL = pluginContext.getHttpTestURL();
        if (pluginContext.urlPassed())
        {
            DataCollectionFloatItem waitTimeForFirstByteItem = new
DataCollectionFloatItem(System.currentTimeMillis(),
httpTestURL.getResponseHeaderWaitTime());

            simpleSequence.addItem(waitTimeForFirstByteItem);
        }

    }


    [...]


    /**
     * Finalize plug-in at end of load test.
     */
    public void deconstruct(Object context)
    {
        LoadtestPluginContext pluginContext = (LoadtestPluginContext) context;

        simpleSequence.addClusterOption(DataCollectionSequence.
        CLUSTER_OPTION_MERGE_FLOAT_ITEMS_TO_SUM, "Merged Cluster Data");

        pluginContext.getPerformanceData().addDataCollectionSequence(simpleSequence);
    }

    [...]


}   // end of class
```

The following screenshot shows how the additional measurement data would be displayed in the GUI. If more than one Measurement is inserted in the Load Test Result, a drop-down list is provided above the diagram to allow the selection of the desired Measurement. All additional measurement diagrams are also included in the PDF report.



Hint: in this example, instances of **DataCollectionFloatItem** are used to store the values. This means that the X-axis of the diagram is already determined and reflects the point in time when the values have been measured. It is also recommended that you predetermine how the data are merged at cluster level. This can be done by using the method **DataCollectionSequence. addClusterOption(<int option>)**.

As an alternative you can use instances of **DataCollectionYZFloatItem** which allows you to control the unit of the Y-axis as well as the unit of an additional Z-axis. Each instance of **DataCollectionYZFloatItem** contains three dimensions which are:

X-Dimension: time stamp (determined)

Y-Dimension: value no. 1 (discretionary useable)

Z-Dimension: value no. 2 (discretionary useable)

As a consequence of adding instances of **DataCollectionYZFloatItem** to a **DataCollectionSequence,** the GUI can show for each of such a data collection three different diagrams in the load test result: XY-diagram, XZ-diagram, and YZ-diagram. However, only the YZ-diagram shows the correlation between "value no. 1" and "value no. 2".

The **DataCollectionSequence** contains a special constructor to support **DataCollectionYZFloatItem** which allows you to label both axis and also allows you to predefine if the GUI should show all three diagrams or should show the YZ-diagram only.

Example 2: The following more complex example demonstrates how, using a local Plug-In thread, external measurement data can be integrated into a Load Test result. This example Plug-In has the **global** Initialization Scope. The local thread will be started in the Plug-In **construct** method when the Load Test program is started, and will be stopped in the Plug-In **deconstruct** method at the end of the Load Test program. The collection of the measurement data occurs inside the thread's **run()** method. Immediately after the local thread is (probably) stopped, the measurement data is inserted into the Load Test Results. The **deconstruct** method will only wait for a maximum of 10 seconds at the end of the thread in order that the end of the Load Test program not be delayed too long. If the local thread continues to run after this point, it does not matter as the Load Test program itself will end, outside the Plug-In code, via a System.exit() call - and this will cause the Java Virtual Machine running the Load Test Job to exit.

```
[...]
import dfischer.utils.DataCollectionSequence;
import dfischer.utils.DataCollectionFloatItem;


/**
 * Load test add-on module.
 */
public class ExternalMeasuringData extends Thread implements LoadtestPluginInterface
{
    private LogVector logVector = null;

    private PerformanceData performanceData = null;            // load test result
    private Thread dataCollectThread = null;                // data collecting thread
    private DataCollectionSequence dataCollectionSequence = null;  // external data


    [...]


    /**
     * Initialize plug-in at start of load test.
     */
    public void construct(Object context)
    {
        LoadtestPluginContext pluginContext = (LoadtestPluginContext) context;
        performanceData = pluginContext.getPerformanceData();

        // start data collecting thread
        dataCollectThread = new Thread(this);
        dataCollectThread.start();
    }


    /**
     * Thread - used to collect external measuring data
     */
    public void run()
    {
        // create data structure for external data and define GUI diagram settings
        dataCollectionSequence = new DataCollectionSequence(1, "Database Calls", "",
                                                            "Calls per Second", null);

        // collect external measuring data in a loop
        while (!isInterrupted())
        {
            // get external data snapshot
            float externalValue = 10.0f;  //  <<< actual value of external data
                                          //  [ add your own code here to
                                          //    accumulate external data ]

            // add external data snapshot to data collection
            DataCollectionFloatItem dataItem =
             new DataCollectionFloatItem(System.currentTimeMillis(), externalValue);
            dataCollectionSequence.addItem(dataItem);

            // sleep for one sampling interval. The sampling interval is
```



```
            // arbitrary configurable on the GUI when starting the load test
            try
            {
                sleep(performanceData.getSamplingInterval() * 1000);
            }
            catch (InterruptedException e)
            {
                interrupt();
            }
        }
    }


    /**
     * Execute plug-in at start of load test.
     *
     * Intrinsic plug-in implementation.
     */
    public void execute(Object context)       // no action in this method
    {
    }


    /**
     * Finalize plug-in at end of load test.
     */
    public void deconstruct(Object context)
    {
        // stop data collecting thread
        dataCollectThread.interrupt();
        try
        {
            dataCollectThread.join(10000);
        }
        catch (InterruptedException e) {}

        // add external measuring data to load test result
        performanceData.addDataCollectionSequence(dataCollectionSequence);
    }


    [...]


}   // end of class
```

Note: If the additional measurement data is retrieved solely from a URL Response, or computed from already existing ZebraTester measurement data, a local thread is not required.

3.4.3.7 Sending E-Mails via SMTP

E-Mails can be sent from within a Plug-In by using the **dfischer.utils.SmtpMessage** class. This requires the availability of an SMTP E-Mail Relay Server which accepts the forwarding of such E-Mails. As most E-Mail systems will not allow the sending of E-Mails from anonymous or unauthenticated users (or programs), it is possible to provide an E-Mail username and password.

The below program fragment illustrates how to send an E-Mail in HTML format via an SMTP E-Mail Relay:

```
String htmlMessage = "<body><html><ul><li>first</li><li>second</li></ul>
<font color=\"blue\">blue sea</font></body></html>";

SmtpMessage smtpMessage = new SmtpMessage(smtpHost);
smtpMessage.setSmtpAuthentication(authUsername, authPassword);
smtpMessage.markHtmlMessage();

smtpMessage.setDebug();
smtpMessage.send(from, to, subject, htmlMessage);

System.out.println("--- waiting for completion ---");
smtpMessage.waitForSendCompletion();

// debug message transfer
String[] debugOutput = smtpMessage.getDebugOutput();
for (int x = 0; x < debugOutput.length; x++)
  System.out.println(debugOutput[x]);
```

3.4.3.8 Handling Time Zones and Date Computations

Date computations in Plug-Ins can be performed, as customarily done in Java, by using the **java.util.GregorianCalendar** class. This must be initialized with the Time Zone currently configured in ZebraTester in order that the computations are correct. Use the class **dfischer.utils.ZoneTime.getGregorianCalendar()** to get the current date in the ZebaTester Time Zone.

Example:

```
[...]
import java.util.Calendar;
import java.util.GregorianCalendar;
import dfischer.utils.ZoneTime;

[...]
// get current date and time in configured ZebraTester time zone
GregorianCalendar cal = ZoneTime.getGregorianCalendar();

// add 3 days to current date
cal.add(Calendar.DAY_OF_YEAR, 3);

// get future date
int futureDay = cal.get(Calendar.DAY_OF_MONTH);   // value range 1..31
int futureMonth = cal.get(Calendar.MONTH);        // value range 0..11, 0 = January
int futureYear = cal.get(Calendar.YEAR);
```


3.4.3.9 Defining and Releasing Own Types of Errors

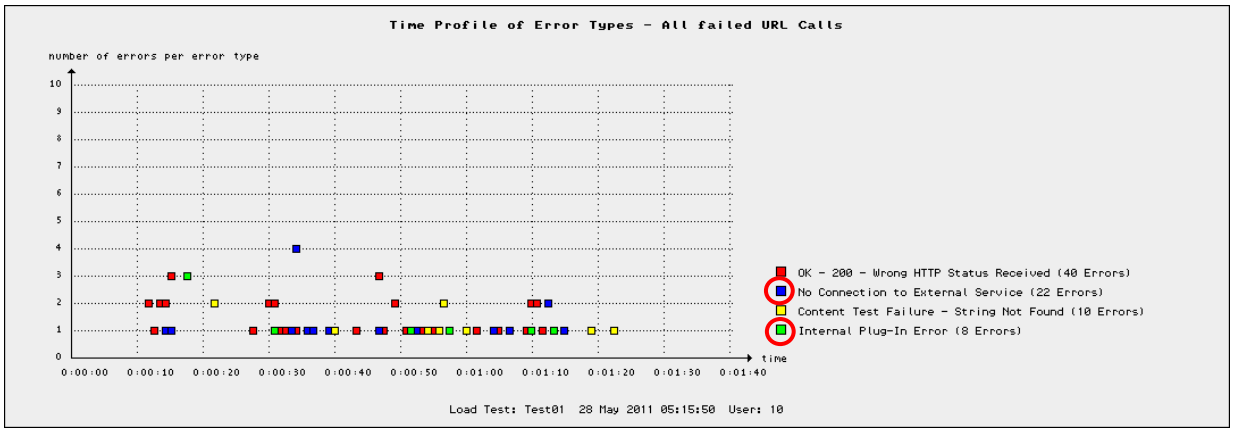
Plug-ins, which are bound to one or to all URLs can turn a successful URL call into a "red" error. In such a case the current loop of the simulated user is aborted and the user continues to execute the next loop.

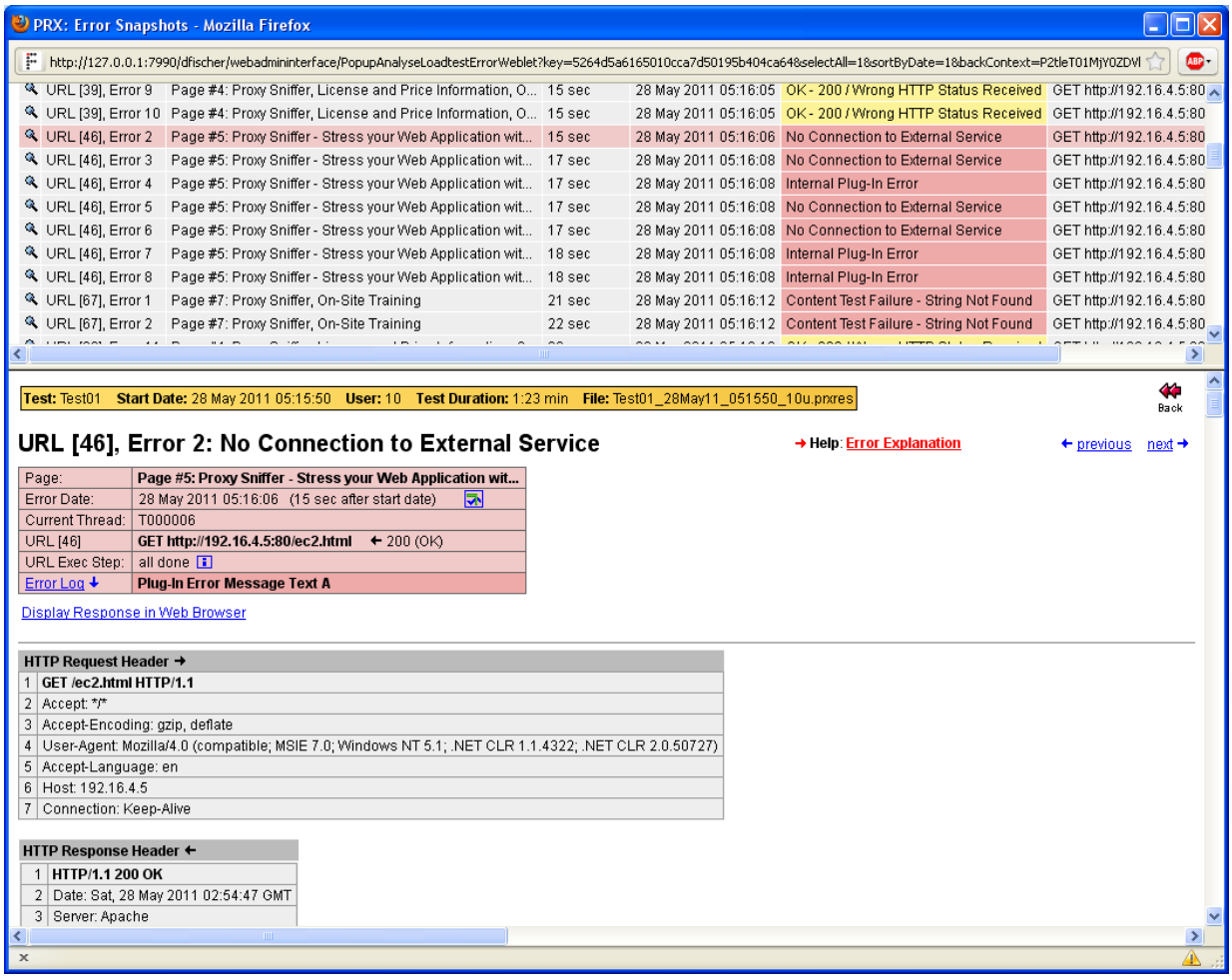
Example:

```
/**
 * Execute plug-in after URL call.
 *
 * Intrinsic plug-in implementation.
 */
public void execute(Object context)
{
        logVector = new LogVector();
        LoadtestPluginContext pluginContext = (LoadtestPluginContext) context;

        if (firstCall)
        {
                // definition of plug-in specific error types

        pluginContext.getPerformanceData().setErrorStatusTypeTranslation(HttpTestURL
.STATUS_TYPE_PLUGIN_ERROR_CODE_1, "No Connection to External Service");

        pluginContext.getPerformanceData().setErrorStatusTypeTranslation(HttpTestURL
.STATUS_TYPE_PLUGIN_ERROR_CODE_7, "Internal Plug-In Error");

                firstCall = false;
        }

        if (...)
        {
                // report plug-in specific "red" error and abort current loop of
                // simulated user -> continue with next loop of simulated user

        pluginContext.markUrlAsFailed(HttpTestURL.STATUS_TYPE_PLUGIN_ERROR_CODE_1,
"Plug-In Error Message Text A");
        }

        if (...)
        {
                // report plug-in specific "red" error and abort current loop of
                // simulated user -> continue with next loop of simulated user

        pluginContext.markUrlAsFailed(HttpTestURL.STATUS_TYPE_PLUGIN_ERROR_CODE_7,
"Plug-In Error Message Text B");
        }
}
```

Up to 10 different error types can be defined by a plug-in (STATUS_TYPE_PLUGIN_ERROR_CODE_0 - STATUS_TYPE_PLUGIN_ERROR_CODE_9) and each released error can contain an own text description about the error.

The plug-in specific errors are shown in the "normal" charts and tables of the load test result details (together with all other measured errors):





3.4.4 Plug-In Programming for High Performance

The execution time of the Plug-In itself does not influence the measured response times; however, the Plug-In will use CPU resources on the local machine, and this can reduce the number of virtual users which can be simulated by the Exec Agent (because the CPU will become overloaded earlier). Refer to the document "Guide to the Successful Execution of Load Tests", Chapter 4, point 4 for more information.

3.4.4.1 Pre-computing Results

In order to reduce the CPU overhead of a Plug-In as much as possible, all repetitive code (code which returns the same value for repeated calls to the **execute** method) should be designed to compute the return value only in the first run, and store the result locally in the Plug-In for subsequent runs. Later calls to this code can simply return the locally-stored value and avoid re-computing the result.

In order to do this kind of optimization without a large effort, the **java.util.HashMap** class is ideal. The missing internal synchronization in this class is not a hindrance, as the execution of the Plug-In will be synchronized by ZebraTester itself..

Example:
For each URL Call, the Web Page (text of the Page Break) must also be determined.

The non-optimized code looks like this:

```
public void execute(Object context)
{
    LoadtestPluginContext pluginContext = (LoadtestPluginContext) context;
    PerformanceData performanceData = pluginContext.getPerformanceData();
    int threadStep = pluginContext.getThreadStep();

    String pageName = performanceData.getPageInfoTextOfUrl(threadStep);

    [...]
```

In the above code, the text of the Page Break is computed each time using the current URL index (threadStep). This indicates that in ZebraTester only the writing and collection of measurement data is already CPU-optimized, not the reading of measurement results.

To optimize the code, the Page Break is retrieved only once, and stored in a HashMap. The optimized code looks like this:

```
import java.util.HashMap;

public class TestPlugin implements LoadtestPluginInterface
{
    HashMap pageInfoMap = new HashMap();

    [...]

    public void execute(Object context)
    {
        LoadtestPluginContext pluginContext = (LoadtestPluginContext) context;
        PerformanceData performanceData = pluginContext.getPerformanceData();
        int threadStep = pluginContext.getThreadStep();

        String pageName;
        Object o = pageInfoMap.get(new Integer(threadStep));
        If (o != null)
           pageName = (String) o;     // Use the stored value
        else
        {
           // Compute the result the first time and store it for later use
            pageName = performanceData.getPageInfoTextOfUrl(threadStep);
```



```
            pageInfoMap.put(new Integer(threadStep), pageName);
        }

        [...]
```

This is only an example. This type of optimization is possible in many programming cases which involve repetitively-executed code, and where the result can be pre-determined. It is worth it to consider which values can be computed beforehand, and which cannot.

3.4.4.2 Disk and Network I/O Operations

I/O operations do not usually require much CPU; however, internal operating system processes involve asynchronous Events which - depending on the volume of I/O operations - can make the entire operating system slower, and cause almost all operations to require a longer time to complete individual system routines.

In order to optimize I/O operations in Plug-Ins, the **execute** method should avoid the frequent opening and closing of I/O connections. The connection should be opened only in the first call to **execute** - or alternatively in the **construct** method. After the connection is opened, all I/O operations should use this open connection. Finally, the connection should be closed in the **deconstruct** method.

For disk I/O operations which create a file during a Load Test, it can be useful to store all file data during the Load Test in memory (e.g. using **java.util.ArrayList** or **java.io.PrintWriter(java.io.ByteArrayOutputStream)**), and then write the file out to disk in the **deconstruct** method. Note that this would only be advisable if the data is not too large. A rule of thumb is that a Plug-In should not store more than approximately 50 MB in its local memory.

3.5 Example Plug-Ins

Examples (*.java source code) of ready-to-use Plug-Ins can be found in the **Project Navigator Directory "MyTests \ Plugins**".

Note: When ZebraTester is re-installed, these ready-to-use Plug-Ins are overwritten. If the Plug-In code has been modified, save the modified ".java" files under different names in the "**MyTests** / **Plugins**" directory - and do not forget to make appropriate adjustments to the Plug-In class names in the source code.*

4 Manufacturer

Ingenieurbüro David Fischer AG, Switzerland | A company of the Apica Group

Product Web Site: http://www.zebratester.com

Apica AB: http://www.apicasystem.com