testPro

The Testing Professionals

TAF PRO

Java Scripting Guide

Version 5.2

DISCLAIMER

The authors have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No Liability is assumed for incidental or consequential damages in connection with or arising out of the use the information or instructions contained herein.

CONTENTS

1 Introduction

This ***TestPro Automation Framework*** (TAFPro) Scripter Guide for Java has been written to assist those who write Java Test Scripts for TAF and use popular third party testing tools such as Selenium.

The TAFPro framework embodies TestPro's approach to automated testing processes and methodologies:

- TAFPro is a hybrid automated testing framework that combines elements of data-driven and keyword-driven frameworks and adopting a role-ring of automation scripts to improve efficiency and scalability;
- TAFPro reduces the test maintenance overhead;
- TAFPro provides a non-technical user interface for Business Users and Testers;
- TAFPro captures and leverages the domain knowledge of business users into a set of regression tests that can be executed in a repetition-based approach to test development and execution.

The key objective of an automation framework is to enable reuse and process variation without needing the business users to be present.

The key functions within TAFPro are:

- Execution of Java tests built to use third party tools;
- Easing data handling challenges for data being used during testing;
- The linking of tests to form scenarios and suites of associated tests.

TAF Pro has been developed with a focus on:

- Minimising the impact of the framework on script development;
- Approaching test execution from a business perspective (as opposed to purely testing);
- Providing flexibility in data association and selection.

1.1 Overview

The key aspects of the TestPro Automation Framework architecture and implementation approach are:

- Review
  - applications to be tested and test objectives
  - type of test user interface required e.g. Spreadsheet, or GUI/database driven
  - automation tools to be used
- Define an automation framework architecture that is *efficient, scalable and maintainable*
- Implement a library of 'adapter' test scripts that test specific functions in the application(s) under test. These scripts are implemented using the automation tool selected by the client.
  - These adapter scripts reduce test maintenance because when the application under tests changes, the changes are typically accommodated within the library of adapter scripts, so that most of the functional tests will not require changes.
- Provide a simple, non-technical user interface to the function library so that business users can develop tests. Typically this interface will be Spreadsheet, or GUI/database driven

1.2 TAFPro Java Fundamentals

1.2.1 RFT Test Scripts vs Java Test Scripts

The original version of TAFPro was designed to work with IBM Rational Functional Tester (RFT), an application test automation tool. RFT provides Test Object capturing and recognition, and uses Java as the underlying test scripting language to operate upon those Test Objects of the application under test. The new version of TAFPro allows test scripts to be written and executed without the need to install RFT. Object recognition services are provided by the third party tools such as Selenium. Java is still used as the test scripting language.

In order to differentiate between the two types of Java test scripting environments, we will call test scripts written for the RFT environment, **RFT Test Scripts**, and test scripts written for the non-RFT environment, **Java Test Scripts**.

1.2.2 Java JAR version differences

The TAFPro solution is based on Java, as are the Java based tools that are integrated with it. In some instances the version of a JAR used by TAFPro may be a different version to that used by a tool. When using two tools together such as Selenium and SeeTest, both of these tools use log4J internally as does TAFPro itself.

In the first instance, the diagnostic strategy should be to identify which is the latest version of the JAR in question and manage the build path (classpath) in TAFPro so that only that latest version appears.

1.2.3 TafJavaScriptLoader

The instance of the TafJavaScriptLoader represents the execution of the TAF Suite. Each instance of the test class (that extends JavaScriptSuperClass) is instantiated in response to a row of applicable test data. This class allows Java Test Scripts utilise IDE feature such as debugging while using TAF test management and control.

1.2.4 JavaScriptSuperClass

TAFPro architecture utilises a superclass that the test classes need to extend. Java variables and classes that are declared in a test class have the scope of that class. In order to declare variables and methods that will be available across all your tests, place these in “JavaScriptSuperClass.java”.

There are number of options to set variables and class modifiers. One good practice is to set them “static” as much as possible in order to prevent unnecessary instantiation of objects in different test classes.

1.3 Using the TAF Test Script Template

TAFPro test scripts include information that is read by TAFPro at the time the script is registered in TAFPro. Detailed instructions for registering scripts is provided in the *TAFPro User Guide*.

1.3.1 TAFPro Java Test Script Template

The Java Test Script template by default contains the name of tafIntegration package. This allows the template to sit in the tafIntegration source folder and not generate IDE warnings. When the template is copied the package name must be changed.

A TAFPro Java Test Script is the Java code that you write in the execute() method between the try/catch. There is one class per test, and every class extends the JavaScriptSuperClass that gives access to all the superclass methods and TAFPro Framework behaviour. Note that the default exception handling is provided by the TAFPro framework via the logStoreException() method.

```
package <package name>;

public class <test name> extends tafIntegration.JavaScriptSuperClass {
	/**
	 * @testpro app="" action="" description=""
	 * @since
	 * @author
	 */
	public void execute() {
		try {

		} catch (Exception ex) {

			logStoreException(ex);

		} finally {

		}
	}
}
```

1.3.2 TAFPro Script Header

Registered scripts contain actions that can be used in TAFPro Execution Suites. TAFPro identifies actions in Java classes by parsing the information present in the script header. The figure below shows a script header template.

```
	/**
	 * @testpro app="" action="" description=""
	 * @since
	 * @author
	 */
```

The “app” and “action” fields are used to identify the test script when preparing test suites inside TAFPro.

The script header is found in the TAF Test Script Template, and should be copied from the “JavaTestScriptTemplate.java” file located in project “src/tafIntegration” folder. Instructions for installing “tafIntegration” folder is explained in section 2.1.

Note that the template script header format should not be modified. Code clean-up features in some Java IDEs could break a long @testpro line for the sake of readability. Attention should be drawn to keeping script header format intact on a single line.

1.4 Creating an Environment in TAFPro

When creating a new non-RFT environment in TAFPro ensure that the environment type is set to JAVA. This means that the TAFPro runtime environment will be generated using the superclasses appropriate to Java Tests Scripts and not RFT Test Scripts. It also means that when scripts are registered, the java source will be parsed for the getData() method call and not dpString() method calls as used in RFT.

TAFPro environments are configured by specifying four main paths at login as follows:

Path Name	RFT Usage	Java Usage
Data Store Path	Root of RFT project	Root of compiled packages, e.g. <project>/bin
Output Files Path	Folder containing csv and xls files for datagroup import/export	As per RFT Usage
Log File Path	Folder where execution/error logs are written	As per RFT Usage
Test Program Path	Path to core RFT jar	Not used. Can be left blank, or set to c:\

1.5 TAFPro Execution Modes

There are four execution modes available when using TAFPro. These are:

- Standalone IDE Debugging
- TAF Debugging
- TAF Execution
- CLI Execution

1.5.1 Standalone IDE Debugging

In standalone IDE debugging you use the native debugging features of the IDE of your choice to run the execute() method that contains your Java Test Script. When using unit test data in a CSV file, define a dummy method that calls executeCsv("path-to-csv"), this in turn causes your execute() method to be called with getData() methods sourcing data from the CSV file.

In standalone mode there is no connection to the TAFPro application or database. Your Java Test Scripts do not need to be registered and no TAFPro environment configuration is required.

1.5.2 TAF Debugging

In TAF debugging mode you can use the debugging (breakpoints etc) features of the IDE of your choice while the Java Test Scripts are being executed by TAFPro and being supplied with data from the TAFPro database. To do this your scripts need to be registered, an execution suite needs to be configured in TAFPro and the associated data groups must be created and populated with test data.

To allow connection to the TAFPro database during debugging the following two files need to be copied into your IDE project at a location that will be on the classpath. We recommend the data store path, which is the root of the compiled packages. In Eclipse this would be <project>/bin.

- hibernate.properties
- Hibernate_ext.properties

These two files are found in the TAFPro installation folder, for example:

"C:\Program Files (x86)\TestPro\TestPro Automation Framework"

These properties files are created when TAFPro is installed and are valid only after a database connection has been configured for TAFPro. Copy them into the "<project>/bin" or equivalent depending on your IDE.

To begin debugging open one of your Java Test Scripts and set a breakpoint at a line of interest.

In your IDE, open the file "TafJavaScriptLoader.java" that is found in the tafIntegration source folder of your project.

WARNING: DO NOT CHANGE ANY OF THE SOURCE IN THIS FILE. If you damage the file, copy it again from the TAFPro installation folder.

NOTE: It is recommended that you close any other instances of TAFPro that are already running.

With the "TafJavaScriptLoader.java" file open in your IDE, start the debugger. This java source file contains the **public static void main(String[] args)** method and so this is where the debugger will begin execution.

There is no need to set up any execution arguments in a run configuration.

Once the debugger has begun, it will instantiate the TAFPro framework and the TAFPro application will start. Log in to this TAFPro application with your normal credentials and select the environment that contains your test configuration. Select the suite that contains your scenarios and test cases which encapsulate the Java Test Scripts intended for debugging.

Run the suite.

When a Java Test Script is executed that contains a breakpoint the IDE will stop execution and display your Java Test Script source. Note that data is being delivered to the getData() methods from TAFPro. isFirstLine() and isLastLine() methods also work and deliver true or false correctly.

Depending on the number of rows of data in the data groups of your scenario at the time of execution your Java Test Scripts may be called many times.

At the conclusion of the test run close the TAFPro application to end the debugging session. Your results have been recorded in the TAFPro database and are available for inspection.

1.5.3 TAF Execution

Open the TAFPro application and log in to your selected environment. Configure a suite that contains the scenarios and Java Test Scripts that have been previously registered. Create and populate data groups as appropriate with test data.

Run the suite.

Results are written to the TAFPro database and are available for inspection after the run has completed.

1.5.4 CLI Execution

CLI or "Command Line Execution" is a feature of TAFPro that uses Ant (from Apache Software Foundation) to control TAFPro suite execution and is callable from Windows batch files. After execution results are available in the TAFPro database.

1.6 Registering Java Scripts in TAFPro

When a script is registered in TAFPro the @TestPro app and action data is read, as well as the names of the data group elements referenced in getData() methods. It is on the basis of these names that the datagroup template is created with columns of matching names.

In TAFPro Java environments the Data Store path for the environment points to the head of the compiled packages. For example, when using eclipse this would be the <project>/bin folder.

Note that when registering Java Test Scripts in TAFPro, the Windows file explorer dialog used to identify the java files begins at the data store path. Navigate this dialog back up the project level and then down into the /src folder to find the java files associated with your tests.

2 Java IDEs

Java IDEs (Integrated Development Environments) are software applications enabling Java programmers to edit, debug and run their Java programs. There are a number of different Java IDEs available on the internet. Nearly all of the favourite Java IDEs provide auto code completion, colourful syntax recognition, code reformatting and immediate code error checking. However, there are some key differences between IDEs.

TAFPro has been tested successfully in integration with the four most popular Java IDEs:

- Eclipse
- IntelliJ IDEA
- JDeveloper
- NetBeans

TAFPro project installation procedure is a similar process between all of the above Java IDEs, although there are small differences in configuration. In the following sections we explain the steps required to create and build Java projects integrated with TAFPro for each of the four Java IDEs.

2.1 Install tafIntegration

The TAFPro 5.2 package contains prerequisite files that establish the integration between a user's Java Test Scripts and TAFPro.

The following Java files are required for every Java project:

TafJavaScriptLoader.java	This is the java test script loader and must be included in every project.
JavaScriptSuperClass.java	This is the extension to the TAFPro superclass where suite wide variables and utility classes and methods can be declared.
JavaTestScriptTemplate.java	This is the Java test script template. All Java test scripts should be modelled on this template.

When preparing TAFPro Integration for your IDE project, do the following steps:

1. Locate the project source (src) folder in your IDE
2. Create a new package (subfolder) and name it "tafIntegration"
3. Locate the TAFPro installation folder (i.e. C:\Program Files (x86)\TestPro\TestPro Automation Framework).
4. Locate the 'deploy' folder (i.e. C:\Program Files (x86)\TestPro\TestPro Automation Framework\deploy), and unzip the file tafintegration.zip.
5. Copy the following files to the tafIntegration folder in your project
    a. JavaScriptSuperClass.java
    b. JavaTestScriptTemplate.java

c. TafJavaScriptLoader.java

2.2 Adding Required Libraries

TAFPro Java libraries contain user APIs that enable users to integrate Java scripts with TAFPro. TAFPro Java libraries should be included in each IDE Java Project to enable unit testing. The process for configuring TAFPro and third party libraries for each Java IDE can be different depending on the IDE. See the later section for specific details for each IDE.

3 IDE Differences When Setting Up TAFPro Projects

This section highlights the differences when setting up Java test projects between the four most popular Java IDEs.

In Java environment, depending on your preferred choice of IDE, TAFPro Data Store path should be set to the root of the Java project's compiled packages. The following table shows differences across IDEs when setting up the TAFPro Data Store path in the TAFPro environment settings:

	TAFPro Environment Data Store path
Eclipse	<\Project>\bin
IntelliJ IDEA	<\Project>\out\production
JDeveloper	<\Project>\classes
NetBeans	<\Project>\build\classes

3.1 Eclipse

Eclipse is one of the most popular free Java IDEs in the market. It can be downloaded from the following URL:

https://eclipse.org/downloads/

In this section, the steps required to set up a TAFPro Java test project in Eclipse IDE environment are explained:

1. In the Eclipse IDE, create a new Java Project and save the created project.
2. Copy in the tafIntegration folder and files under \src folder.
3. Right click on the created project name, and select Build Path > Configure Build Path …
4. From the tabs, select Libraries
5. Click Add External JARs…
6. Add the TAFPro jar files from the TAF Pro library folder. The TAF Pro library folder is located in the folder where you installed TAF in (i.e. C:\Program Files (x86)\TestPro\TestPro Automation Framework\lib)
7. Close the Java Build Path window.
8. In Eclipse, right click on your project and select new>Java Package.
9. Choose a name for your project package and click on Finish.
10. Right click on your new created package and select new > Class.
11. Select a Name for the class.
12. Open JavaTestScriptsTemplate.java and copy into the body of the new class file.
13. Change the name of the package to your new package name.
14. Change the name of the class from "JavaTestScriptTemplate" to your class name to match your java file name.
15. Complete your script.
16. In the script header comments, ensure that the app, action and description fields are filled.

3.2 IntelliJ IDEA

IntelliJ IDEA is another well-known IDE used by Java developers community. IntelliJ IDEA developed by JetBrains is offered under two different packages; Community (free version) and Ultimate.

https://www.jetbrains.com/idea/

The following steps outline the process required for creating a new Java project in IntelliJ IDEA as well as importing TAFPro dependencies.

1. Create a new Java project in IntelliJ IDEA environment.
2. Copy in the tafIntegration folder and files under \src folder.
3. Right click on the created project name, and select Open Module Settings > Project Settings > Libraries.
4. Add the TAFPro jar files from the TAF Pro library folder. The TAF Pro library folder is located in the folder where you installed TAF in (i.e. C:\Program Files (x86)\TestPro\TestPro Automation Framework\lib).
5. Apply new settings and close the Project Structure Window.
6. Right click on your project \src and select new> Package.
7. Choose a name for your project package and click on Ok.
8. Right click on your new created package and select new > Java Class.
9. Select a Name for the class.
10. Open JavaTestScriptsTemplate.java and copy into the body of the new class file.
11. Change the name of the package to your new package name.
12. Change the name of the class from “JavaTestScriptTemplate” to your class name to match your java file name.
13. Complete your script.
14. In the script header comments, ensure that the app, action and description fields are filled.

3.3 JDeveloper

JDeveloper is a free Java IDE published by Oracle under two different edition series; Studio and Java Edition. Java Edition is a minimum edition for accessing core features of Java. JDeveloper is accessible through:

http://www.oracle.com/technetwork/developer-tools/jdev/downloads/

The following steps explain the procedure for creating a Java project using TAFPro features in JDeveloper environment.

1. Create a new Java Desktop Application in JDeveloper environment.
2. Create a new Java Class which accordingly creates source folders.
3. Copy in the tafIntegration folder and files under \src folder.
4. Right click on the created project name, and select Project Properties... > Libraries and Classpath > Add JAR/Directory.
5. Add the TAFPro jar files from the TAF Pro library folder. The TAF Pro library folder is located in the folder where you installed TAF in (i.e. C:\Program Files (x86)\TestPro\TestPro Automation Framework\lib).
6. Click on Ok to close the Project Properties Window.
7. Open JavaTestScriptsTemplate.java and copy into the body of the new class file.
8. Change the name of the package to your new package name.
9. Change the name of the class from “JavaTestScriptTemplate” to your class name to match your java file name.
10. Complete your script.
11. In the script header comments, ensure that the app, action and description fields are filled.

3.4 NetBeans

Oracle NetBeans is another popular IDE primarily intended to support Java. NetBeans IDE is a free software accessible from:

http://www.oracle.com/technetwork/developer-tools/netbeans/overview/

The following steps explain the procedure for creating a Java project using TAFPro features in NetBeans environment.

1. Create a new Java Application in NetBeans environment.
2. Copy in the tafIntegration folder and files under \src\Java Application Name\ folder.
3. Right click on the Java Application name, and select Properties > Libraries > Add JAR\Folder.
4. Add the TAFPro jar files from the TAF Pro library folder. The TAF Pro library folder is located in the folder where you installed TAF in (i.e. C:\Program Files (x86)\TestPro\TestPro Automation Framework\lib).
5. Apply new settings by clicking on Ok in Project Properties Window.
6. Open JavaTestScriptsTemplate.java and copy into the body of the new class file.
7. Change the name of the package to your new package name.
8. Change the name of the class from “JavaTestScriptTemplate” to your class name to match your java file name.
9. Complete your script.
10. In the script header comments, ensure that the app, action and description fields are filled.

4 TAFPro Test Structure

When using third party tools with TAFPro, two questions arise with respect to driver classes. All three tools that are canvassed later in this guide use drivers that must be instantiated somewhere in the test.

1. Where should you declare your driver object variable?
2. Where should I instantiate (new) the driver class?

In general, TestPro recommends declaring a variable to contain your driver object in the JavaScriptSuperClass extension. This can be declared as **public static**.

The driver object can be instantiated either by an inline initialization of the variable in the JavaScriptSuperClass extension or explicitly in a StartUp test using the isFirstLine() method to restrict this to the initial row of test data.

TestPro recommends the use of StartUp and TearDown tests to perform test tool environment management at the beginning and end of a suite of tests.

4.1 Test Scenario Structure

Within a test scenario create two test scripts in addition to the test scripts that implement the business goals of the tests. These will be the StartUp and TearDown tests.

Place the StartUp test as the first test script in the scenario, and place the TearDown test as the last test in the scenario.

When TAFPro constructs the consolidated data group for the scenario each test in the scenario will be called to manage its data group fragment, once per row. The number of rows in the consolidated data group is the number of rows in the largest data group of all the tests in the scenario. For example, if there is a test with 500 rows in its data group, then in the consolidated data group there will be 500 rows and the data from the other smaller tests will be repeated. In all, each test you have written will be called 500 times. This includes the StartUp and TearDown tests which typically have a single row each in their own data groups.

In order to have the StartUp test only instantiate the driver once at the beginning of the scenario use the isFirstLine() method in an if statement so that instantiation happens only when isFirstLine() is true.

Similarly, surround any driver deconstruction or “end of test” activity in a block in the TearDown test that is protected by an isLastLine().

5 Third party tools

5.1 Selenium

Selenium is a popular browser automation test tool. It can be downloaded from the following URL:

http://www.seleniumhq.org/

Include the paths to the Selenium libraries in the build path property of you IDE. Note that depending on the browser you wish to control you will also have to include the appropriate driver, such as:

- Chromedriver.exe
- IEDriverServer.exe

These are also available in downloadable packages from the Selenium web site. The folder containing the driver (for Chrome or IE) should be included on the system path so that it can be found at runtime.

5.1.1 log4j.properties file

Selenium uses log4j internally. You must provide a default log4j.properties file into your project when you use Selenium to avoid the log4j configuration warnings. This is a common issue with Selenium, in that when unit testing the Java console will contain benign warnings about log4j.

When using Eclipse this properties file can be copied into the root of the project. With other IDE's it may need to be placed with the compiled packages.

```
A sample log4j.properties file is shown below:
# Set root category priority to INFO and its only appender to CONSOLE.
log4j.rootCategory=INFO, CONSOLE
#log4j.rootCategory=INFO, CONSOLE, LOGFILE

# Set the enterprise logger category to FATAL and its only appender to
CONSOLE.
log4j.logger.org.apache.axis.enterprise=FATAL, CONSOLE

# CONSOLE is set to be a ConsoleAppender using a PatternLayout.
log4j.appender.CONSOLE=org.apache.log4j.ConsoleAppender
log4j.appender.CONSOLE.Threshold=INFO
log4j.appender.CONSOLE.layout=org.apache.log4j.PatternLayout
log4j.appender.CONSOLE.layout.ConversionPattern=- %m%n

# LOGFILE is set to be a File appender using a PatternLayout.
log4j.appender.LOGFILE=org.apache.log4j.FileAppender
log4j.appender.LOGFILE.File=axis.log
log4j.appender.LOGFILE.Append=true
log4j.appender.LOGFILE.Threshold=INFO
log4j.appender.LOGFILE.layout=org.apache.log4j.PatternLayout
log4j.appender.LOGFILE.layout.ConversionPattern=%-4r [%t] %-5p %c %x - %m%n
```

5.1.2 Superclass Customisation for Selenium

Selenium driver objects can be instantiated in the "JavaScriptSuperClass" class. TAFPro works by instantiating the users test class once per row of TAFPro test data. This is why it is more efficient to declare the Selenium driver object once in the abstract superclass extension or in a StartUp test so that it is available across the execution of the entire suite.

```
public abstract class JavaScriptSuperClass extends JavaFrameworkSuperClass
{

	public static WebDriver driver = new FirefoxDriver();
// or
	public static WebDriver driver;		// new an instance of the driver
						// in a StartUp test

	…
}
```

Should the user choose to use the Event Management features of Selenium, then the event handler objects and event handling classes should be declared in the superclass extension so that they are available across the TAFPro suite execution.

```
package tafIntegration;

import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.firefox.FirefoxDriver;
import org.openqa.selenium.support.events.EventFiringWebDriver;
import org.openqa.selenium.support.events.WebDriverEventListener;
import org.openqa.selenium.By;

import au.com.testpro.framework.java.superclasses.JavaFrameworkSuperClass;

public abstract class JavaScriptSuperClass extends JavaFrameworkSuperClass
{

	public static WebDriver driver = new FirefoxDriver();
// or
	public static WebDriver driver;		// new an instance of the driver
in
					// a StartUp test

	public static WebEventListener handler;
	public static EventFiringWebDriver eventDriver;

	// Alternatively place the rest of the event handling setup in a
StartUp test

	public JavaScriptSuperClass() {
super();
		eventDriver = new EventFiringWebDriver(driver);
		handler = new WebEventListener();
		eventDriver.register(handler);
	}

	// EventHandler class and methods

	public class WebEventListener implements WebDriverEventListener {

		public void beforeNavigateTo(String url, WebDriver driver) {
			setMessage("Before navigating to: '" + url + "'");
		}

		public void afterNavigateTo(String url, WebDriver driver) {
			setMessage("Navigated to:'" + url + "'");
		}
```

```
            …
      }
…
}
```

5.1.3 Sample Code

```
package Tests;

import org.openqa.selenium.By;
//import org.openqa.selenium.chrome.ChromeDriver;
import org.openqa.selenium.firefox.FirefoxDriver;
//import org.openqa.selenium.ie.InternetExplorerDriver;

public class SelTest1 extends tafIntegration.JavaScriptSuperClass {
      /**
       * @testpro app="FFtest" action="test1" description="a small test"
       * @since
       * @author
       */

      public void execute() {
            try {
                  driver.get("http://www.google.com");
                  s = getData("SearchTerm");
                  …
            }
            …
      }
      …
}
```

5.2 Experitest SeeTest

Experitest is a test automation tool records interactions on mobile devices. It supports a variety of mobile devices including Android, iOS, Windows Phone and BlackBerry. It enables users to interact, record and play with both on mobile devices and emulators. Experitest-generated script can be run in different Java environments such as IBM RFT and Eclipse.

The trial version of the software enables users to access the software for 30 days. SeeTest can be downloaded from the website:

https://experitest.com/download/seetestautomation/

5.2.1 Install the Toolset

Experitest installation package includes some supporting jar libraries that enables users to record and export Experitest interaction with mobile devices as commands implementable in other environments such as RFT, Junit, Selenium and etc. In this guide, we explain the requirements for setting up a Java project integrating TAFPro with experitest SeeTest.

In your IDE project add the following JARs to the build path:

- imageClient.jar
- Junit-4.4.jar
- Mobile-webdriver.jar
- Selenium-api-2.42.2.jar
- Selenium-support-2.44.0.jar

- Ws-commons-util-1.0.2.jar
- Xmlrpc-client-3.1.2.jar
- xmlrpc-common-3.1.2.jar

5.2.2 Superclass Customisation for SeeTest

Similar to the approach for other third party test tools, the preferred approach for using SeeTest Objects is to declare them once in the JavaScriptSuperClass. Therefore, the superclass should be customised in order to include the declarations expected to be constant or declared only once.

```
package tafIntegration;
import com.experitest.client.Client;
import au.com.testpro.framework.java.superclasses.JavaFrameworkSuperClass;

public abstract class JavaScriptSuperClass extends JavaFrameworkSuperClass
{

	protected final String host = "localhost";
	protected final int port = 8889;
	protected final String projectBaseDirectory =
"C:\\workspace\\project1";
	protected static Client client = null;
}
```

Note: The host, port and projectBaseDirectory are used by the Experitest application and might vary in different environments. The values assigned in the script above are examples.

The SeeTest client instantiation should be done in a StartUp test as previously described.

5.2.3 Sample Code

```
package Test;
import static org.junit.Assert.*;
import org.junit.Test;

public class firstNumber extends tafIntegration.JavaScriptSuperClass {
	/**
	 * @testpro app="ExperiTest" action="firstNumber" description="To
firstNumber"
	 * @since
	 * @author
	 */
	@Test
	public void execute() {
		try {
			char number = getData("Number").charAt(0);
			client.click("default", "element 0"+number, 0, 1);
		} catch (Exception ex) {

			logStoreException(ex);

		} finally {

		}
	}
}
```

5.3 AutoIT

AutoIT v3 is a freeware BASIC-like scripting language designed for automating the Windows GUI and general scripting. It uses a combination of simulated keystrokes, mouse movement and window/control manipulation in order to automate tasks in a way not possible or reliable with other languages.

AutoIT V3 is free to download and use.

5.3.1 autoitx4java

AutoIT is packaged with an interface called AutoITX which supports accessing AutoIT functions through COM objects.

AutoITX4Java uses JACOB to access AutoITX through COM and strives to provide a native Java interface while maintaining the simplicity of AutoIT.

Autoitx4java is an opensource Github project that is free to download and use.

5.3.2 Jacob

JACOB is a JAVA-COM Bridge that allows you to call COM Automation components from Java. It uses JNI to make native calls to the COM libraries. JACOB runs on x86 and x64 environments supporting 32 bit and 64 bit JVMs.

JACOB is an opensource SourceForge project that is free to download and use.

5.3.3 Download the Toolset

Download the full toolset from their respective websites:

Tool	URL	What to download
AutoIT3	https://www.autoitscript.com	The FULL installation: autoit-v3-setup.exe
autoitx4java	https://github.com/accessrichard/autoitx4java	Click [Download ZIP]: autoitx4java-master.zip
JACOB	https://sourceforge.net/projects/jacob-project	Click green Download button: jacob-1.18.zip

Open the two zip files and locate the folders that contain the distribution jars.

5.3.4 Install the Toolset

Install AutoIT3 by executing “autoit-v3-setup.exe “ and follow the defaults to install it into “c:\program files (x86)”.

In your IDE project add the following JARs to the build path:

- AutoITX4Java.jar
- jacob.jar

In Windows add the folder that contains “jacob-1.18-x86.dll” to the system path.

In the JavaScriptSuperClass extension add a declaration for the AutoIT driver object.

```
AutoITX autoitx;
```

We recommend the use of a StartUp test where the AutoITX driver object can be instantiated:

```
autoitx = new AutoITX();
```

5.3.5 Sample

```
public void execute() {
      try {
            autoitx.run("notepad", "C:/Windows/System32",
autoitx.SW_SHOWDEFAULT);

      } catch (Exception ex) {
            logStoreException(ex);
      } finally {
      }
}
```

6 Unit Testing with JUnit

There are some differences in the way JUnit is installed and used across IDE's. Some may require separate installation, some may require ticking JUnit as an option at IDE installation time, or you may have to run a JUnit Setup wizard. Follow the JUnit instructions applicable to your IDE.

If you choose another unit testing tool or choose to declare your own public static void main(String[] args) method then TAFPro will continue to work as expected.

6.1 Using JUnit embedded in your Script

There are two main features of JUnit that affect TAFPro Java Test Scripts.

- import org.junit.Test;
- Use the @Test pragma for JUnit above the execute() method

The method that contains the test script is the execute() method. Annotate the execute() method in your script with the @Test pragma. When you run the class in your IDE JUnit will instantiate your class and call the tagged method. This is a handy way to run your script standalone.

See the sample below:

```
package Tests;
import org.junit.Test;

public class test01 extends tafIntegration.JavaScriptSuperClass {
	/**
	 * @testpro app="test" action="test01" description="test description"
	 * @since
	 * @author
	 */
	@Test
	public void execute() {
		try {
			setMessage("Hello, world");
			…
		}
		…
	}
	…
}
```

Where you have specified driver object instantiation to occur in a separate StartUp test, this can be emulated by using the jUnit @Before prama before a dummy method that performs the instantiation. The driver variable is already declared in the JavaScriptSuperClass extension.

```
	@Before
	public void dummy() {
		driver = new FirefoxDriver();
	}
```

6.2 Unit Testing with CSV Test Data

When using CSV test data to support standalone unit testing, add an addition dummy method to your test class. This dummy method is not seen or used by TAFPro during formal test execution and may be left in the test class. It's only purpose is to annotate the path to the CSV file that will supply the single row of unit test data to the getData() calls in the test script.

Create a new dummy method and set the @Test pragma above this method. The method then calls executeCsv("path to CSV"). See the example below. Note the appropriate use of backslashs in Java strings that contains the path.

```
@Test
public void dummy() {
      executeCsv("c:\\project\\data\\testdata.csv");
}
```

The call to executeCsv(String) loads the CSV test data into the TAFPro framework which in turn calls your local execute() method that contains the test script. The calls to getData() can now be satisfied with data from the CSV file.

If the named file path does not exist an exception is generated and calls to getData() return the empty string.

6.3 TAFPro API Differences when Unit Testing

When unit testing standalone the following differences in TAFPro API behaviour must be noted:

- isFirstLine() and isLastLine() always return false
- setOutput() and getOutput() do nothing
- getActionName() will return the empty string

6.3.1 With CSV Test Data

- When using CSV unit test data getData() returns data for the matching column in the CSV file
- If the column cannot be found getData() returns the null string

6.3.2 No CSV Test Data

getData() return the null string

7 TAFPro User APIs

7.1 Passing and Failing Tests

When writing Java Test Scripts the following pass and fail mechanism is used by TAFPro.

PASS	A test that does not explicitly fail will be deemed to have passed. When writing a Java Test Script within a try/catch block, falling out the bottom of the block will effectively pass the test.
FAIL	In order to fail a test, create an exception and include a failure message.

Sample:

```
	public void execute() {
		try {

			// Java Test Script goes here

			// At some point the test should fail!
			If (logical application error) {
				throw new Exceotion("Failed due to app error");
			}

		} catch (Exception ex) {

			logStoreException(ex);

		} finally {

		}
	}
```

7.2 Extending the Functionality of Scripts

The Scripter can use TAFPro API's to accommodate more than one action existing in a single script as shown the following example:

```
	String action = getActionName();
	if (action.equals("Order")) {
		// do order
	} else if (action.equals("Cancel Order")) {
		// do cancel
	} else if (action.equals("Delete Order")) {
		// do delete
	}
```

The Scripter can also use the TAFPro getOutput() and setOutput() to manage precondition data and output data.

7.2.1 tafStore();

Objective: Display row of data sent from TAF Pro.

Reason: Display the current row of data being used in the script execution.

Data are displayed at three levels, the current script, the current test suite row and output (precondition).

Syntax: tafStore();

Example Usage:

```
tafStore();
```

There are no parameters. The data names and values currently used by TAF Pro will be written to the Rational Functional Tester log.

Example Output for TAF Pro store:

```
[ScriptData={CommonKeyword:Common},{Risk:Medium},{var1:value1}]
[SuiteData={CommonKeyword:Common},{Risk:Low},{var2:value2},{var3:value3}]
[Precondition Data={var1:V12},{var3:V312},{var2:V222}]
```

7.2.2 setMessage("result message");

Objective: Send an information message to the TAFPro result set.

Reason: Setting a message in the TAFPro result set to give more information about the execution result. When used standalone the message is written to the Java console.

Syntax: setMessage(“result message”);

Example Usage:

```
setMessage("Client David successfully created ");
```

Note - setMessage() can be invoked multiple times in a script. The message(s) can be viewed via the Results window at the DataGroup level when execution is complete.

7.2.3 setWarning("warning message");

Objective: Send a warning message in the TAFPro result set.

Reason: Sets a warning message to give more information about the execution result; the script has not passed but did not fail either. When used standalone the message is written to the Java console.

Syntax: setWarning(“warning message”);

Example Usage:

```
setWarning("Expected amount $9.98, actual is $10.00");
```

Note - setWarning() can be invoked multiple times in a script. The warning message(s) can be viewed via the Results window at the DataGroup level when execution is complete.

7.2.4 getData("ColumnNameInSpreadsheet");

Objective: Used for getting the value from the TAFPro data group.

Reason: For data-driven script purposes, the getData() method is used for getting column value from TAF Pro Spreadsheet and as an input value for text field on the application.

Syntax: getData("ColumnNameInTestData");

Returns: Returns the value of the column of test data for the column named in the parameter. Creates an exception where the named column does not exists in the current row of test data.

Example Usage:

```
clientText().setText(getData("ClientName"));
```

Note: The name ***must*** be a quoted string; it cannot be an expression or a variable.

7.2.5 getActionName()

Objective: Get the action value from data group

Reason: This method returns a string value from the supplied data group that can control the processing flow of the active script

Syntax: getActionName();

Returns: Returns the name of the current action in TAF. When used standalone getActionName() returns the empty string.

Example Usage:

```
if (getActionName().equalsIgnoreCase("Create")) {
      //  Do CREATE
} else {
      //  Do MODIFY
}
```

7.2.6 isFirstLine()

Objective: To be able to identify if the current row of data is the first row.

Reason: This method provides the ability to do conditional processing based on the first row and to avoid processing if testing is on additional data rows. In using this method you should also

refer to Scenario Properties in the TAF Pro User Guide for the Iterate capability within TAF Pro.

Syntax: isFirstLine();

Returns: True if the current row of data is the last row in the suite, False otherwise. When standalone during unit testing isFirstLine() always returns False.

Example Usage:

```
if (isFirstLine()) {
// Do first line processing
} else {
	// Do alternative processing
}
```

7.2.7 isLastLine()

Objective: To be able to identify if the current row of data is the last row.

Reason: This method provides the ability to do conditional processing based on the last row and to avoid processing if testing has not reached the last line of data. In using this method you should also refer to Scenario Properties in the TAF Pro User Guide for the Iterate capability within TAF Pro.

Syntax: isLastLine();

Returns: True if the current row of data is the last row in the suite, False otherwise. When standalone during unit testing isLastLine() always returns False.

Example Usage:

```
if (isLastLine()) {
	// Do last line processing
} else {
	// Do alternative processing
}
```

7.2.8 executeCsv()

Objective: Nominate the path to a CSV file to be used during standalone unit testing.

Reason: This method provides the ability to supply the path to a CSV file that contain two lines. The first is the line that contains the comma separated headings, the second contains the comma separated data.

Syntax: executeCsv();

Returns: void

Example Usage:

```
executeCsv("c:\\project\\data\\testdata.csv");
```

7.2.9 Finally{}

Objective: Defaulted as part of the TAF Pro template for all scripts created in a TAF Pro Enable project. Finally provides for any processing as the last step in the script regardless of the script state.

Reason: Finally should contain methods that will be executed regardless of whether the script has thrown an exception or not.

Syntax: finally { … your code ... }

Example Usage:

```
try {

// script code

} catch (Exception ex) {
// logToolException(ex);
} finally {
      // free memory between tests
      unregisterAll();
}
```

7.3 Precondition Data

TAF Pro variables that implement precondition data may be passed from one script to another via setOutput and getOutput methods.

Variable values are retained for ***each row of data*** executed and are current for every action executed in every test case in every scenario of the current row. When TAF Pro begins execution of the next row, all variable values are erased and new values created. Variables are **NOT** passed from one row of data to another.

An alternative would be to declare a public static variable in the superclass extension.

7.3.1 setOutput("OutPutName", outPutValue);

Objective: Used for setting precondition data.

Reason: Keep execution results in TAF Pro, so other scripts can use those results as input data to satisfy some processing flow.

- The maximum length of the message is 767 characters.

Syntax: setOutput("OutPutName", outPutValue);

Example Usage:

```
setOutput("Created Client", ”David”);
```

7.3.2 getOutput("OutPutName");

Objective: Used for getting precondition data.

Reason: Getting precondition data as an input data for other scripts.

Syntax: getOutput("OutPutName");

Example Usage:

```
MatterManagement().inputKeys(getOutput("CreatedClient"));
```