Zenith Automation Framework

Table of contents

Learning Scripting with TestComplete and JScript

JavaScript Tutorials

If you are not familiar with JavaScript the best place to start is to work through the video tutorials listed below.

General Notes

Not all tutes in the series are relevant. JavaScript is a web page language any tutes that relate to manipulating web pages are not directly relevant to its use in test automation.

Instead of using a web page for the tutorials and document.write to read the output of your application create a new TestComplete suite, add a JScript project and add the following code to unit1:

```
// Use this instead of document.write
// don't bother putting any html like <br/>
// into these messages they are used because
// the tutorial writes to html
function log(message){
 Log.Message(message, message);
}


function tuteFunction(){
  // your code goes here
}
```

Tute 1

Basics

Contents

- variables
- string functions
- types and typecasting
- comparing values
- logical operators
- array action
- conditional operators
- looping
- functions

Notes

Normal Variables:
underscores only used in special cases you will learn about later
always starts with lowercase letter (camelcase)
e.g. var loanAmount = 25000;

Constants
only upper-case and underscore
e.g. var CONVERSION_RATE = 1.23;

Escaped Characters utility function newLine() is used instead of \n in the framework

Comparing Values

Use === and !== instead of != and ==

For in

use the for statement instead or _.each (available in utility functions in framework)

Functions

use the same script file for your functions ignore the info about adding an external script to a page

Tute 4

Arrays and Functions

Contents

- array functions
- multi dimensional array
- functions arguments
- arguments

Tute 8

Exceptions

Notes

Use variables instead of a form
In the framework use the utility functions throwEx and ensure

TestComplete Tutorials

Debugging with TestComplete

Finally

A longer video that goes beyond the scope of what is required but highly recommended for those who want a deeper understanding of JavaScript: Link

Users' Guide

UI Testing in Context

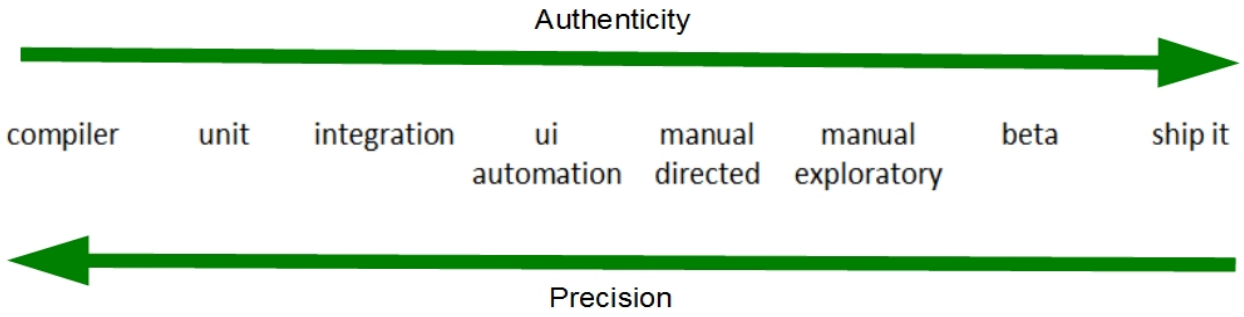
The Zenith Automation Framework is a functional UI testing framework which can also be used for batch and REST service testing. Before delving into the details of the this framework it is important to understand the role UI testing in the broader context of an organisation's QA stack. UI based solutions can be a powerful addition to an organisation's QA toolkit but in order to deliver value these solutions need to be implemented correctly and with full understanding of the strengths and limitations of such an approach.

The Precision / Authenticity Continuum

When considering development of an application a number of points can be identified at which the functions within an application are put to the test. These include:

- Compiler validation and warnings
- Unit tests
- Integration tests
- Automated UI tests
- Manual Directed tests
- Manual Exploratory tests
- Beta releases
- Product release

This list can be read as a continuum from small highly precise tests that only exercise a specific fragment of functionality through to full stack tests that realistically stress the whole application. The final test is the product release.

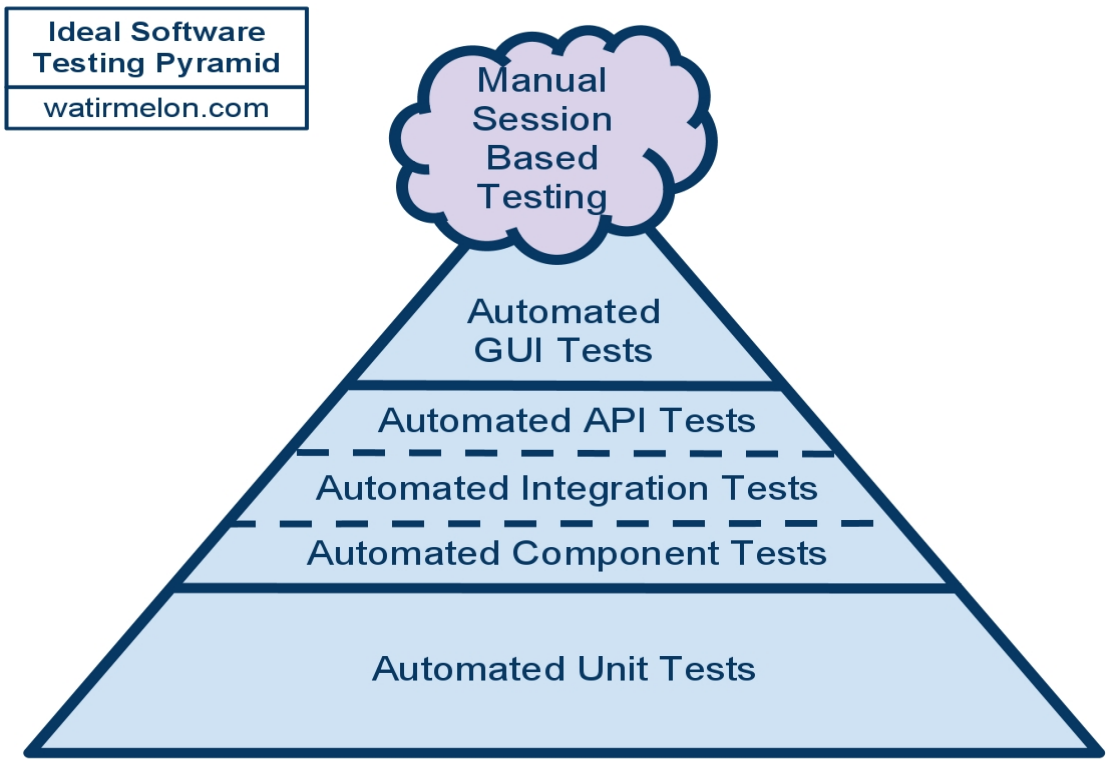


A number of observations can be made about the properties of these tests:

1. Ease and speed of execution is greater for higher precision tests. Unit tests run extremely quickly and the test environment requires little preparation. Full stack testing, by contrast, whether this be manual or automated UI tests require, a full environment setup and tear down and may include the deployment of servers and databases. Full stack tests are magnitudes slower than unit tests and integration tests. Manual tests are magnitudes slower and more expensive to run that automated full stack tests.

2. High precision tests are more useful to developers than low precision tests. When a low precision test such as UI automation test fails then it usually requires further analysis by QA and a developer before the underlying cause is identified. Unit tests, by contrast, will fail at the point in the code of the defect.
3. The probability of finding defects that are incidental to the feature under test are higher with more authentic, less precise tests such as automated UI tests and manual tests.
4. Type 2 errors, i.e. test errors that are due to errors in the test rather than defects in the application, are more prevalent in low precision tests. This is particularly the case with UI based automation tests because a full stack test is being run by an automation framework and consequently does not have the benefit of a person to discriminate between what are acceptable changes in application behaviour and application defects.
5. More authentic tests such as UI automation and manual exploratory tests can test a feature or a number of features as a whole. This can expose defects not detectable by lower level tests.
6. The cost of a defect being found increases the less precise a test is. Compiler warnings and errors are immediate and can be fixed straight away. Unit tests can be fixed quickly because they fail at the site of a defect and are generally run high frequency because of the ease and speed with which they can be run. Failures in automated UI tests and manual UI tests usually happen later in the development process, often require considerable analysis and the use of a defect tracking system for the defect to be addressed. Further along the continuum a defect encountered in production is magnitudes more expensive to isolate and fix and can also involve loss of good will and cost of redeployment / release.

From the above an overall picture emerges of high precision tests being cheap, easy to develop and run and easy to fix when defects are encountered. Low precision tests can test more holistic functionality and provide wider test coverage per test. They are, however, slower and more expensive to execute and require additional analysis and management when defects are encountered. From these insights it makes sense to have a lot of high precision tests and a smaller number of low precision tests to test features more holistically and provide a validation layer for the lower level tests. This idea is summarised by the concept of a test pyramid. A Google search will throw up many variants of this idea. One such illustration of the test pyramid is shown below.



Source

Separation of Concerns

When considering the role of UI automation tests the fact that UI automation can (and should) be conducted by QA gives UI automation extra value as a control. If Development takes responsibility for higher precision tests and QA develops and executes higher level UI automation tests then lapses by Development should be detected relatively early due to UI automation test failures. If Development is responsible for UI tests then a break down in process within the development team is likely to result in UI tests not being maintained hence losing their value as a control.

Generally developers should not be maintaining UI automation tests. Their time is far more effectively spent on lower level tests and ensuring an architecture is maintained that enables such tests to be run. UI automation should be implemented by adding the necessary basic scripting / coding skills to QA not subtracting resources from Development.

Architectural Constraints

Another area where automated UI tests may be emphasised more is when dealing with poorly factored legacy code. High precision tests cannot be written against a code base that is a monolithic ball of mud and The Big Ball of Mud is a popular design paradigm in many legacy systems. Refactoring such systems so as to enable high precision tests can be risky. A common approach to mitigating this risk is identify high value areas of the application that are expected to undergo change and to start by creating a comprehensive set of automated UI regression tests across these areas. With this safety net in place developers are able to more confidently refactor the code base to enable high precision tests and higher quality development.

Introduction to the Zenith Automation Framework

The Zenith Framework is a collection of JScript (Microsoft's version of JavaScript) utilities and conventions that provide a solution to most of the common problems encountered when creating a suite of UI or REST automation tests. Every automation project will present different challenges but usually many of the underlying requirements are actually very similar, if not the same. Functions such as waiting for UI objects, running different test suites, refreshing the test environment and defining data driven tests are examples of such generic requirements.

The Zenith Framework is built with and runs in TestComplete. Many of the utility functions provided are simple wrappers around TestComplete functions. The services provided by the framework such as run time test selection and data driven testing can be achieved in different ways using TestComplete without the framework. The Zenith Framework, however, aims to simplify commonly performed automation tasks to an even greater extent than TestComplete does on its own. It also extends the functionality of TestComplete and provides a template that guides the user toward development practices which ultimately leads to reduced effort required for development and maintenance.

TestComplete is a mature automation tool having been first released in 1999 and is still being actively developed and improved by SmartBear. The test development environment and documentation that comes with TestComplete is second to none. TestComplete forums also provide a rich source of useful information and when you come up against problems that you can't solve, SmartBear support is highly responsive. Although TestComplete is a commercial product, the license costs can be kept under control by matching the modules purchased closely to your requirements.

The Zenith Framework is script based. This reflects the firmly held belief that non-script based tools should be avoided. Alternatives such as record and play back, GUI based facades, keyword or spreadsheet driven frameworks share the characteristics of being good for small problems and sales demonstrations but being highly inflexible and extremely difficult to maintain. This can lead to a dependence on developers or external consultants and/or the abandonment of the automation. The use of the Zenith Framework requires QA to learn scripting. The fact that the framework relieves consultants from the need to write generic functions and

services means that more time throughout an engagement can be allocated to training. QA staff can be taught the fundamentals of scripting and how to avoid the mistakes that frequently undermine automation efforts. The concepts taught will be the basic principles of good software development.

UI automation is a software development practice. The Zenith Framework embraces the understanding that UI automation is a software development by codifying fundamental software development principles within the framework.

Getting Started

Developing any UI automation is an exercise in software development and the Zenith Framework is a script based TestComplete framework. This being the case, becoming proficient in using this framework for developing UI automation tests has the following prerequisites:

1. Basic JScript skills
2. Proficiency in using TestComplete
3. An understanding of the conventions, structure and functionality of the framework
4. An understanding of fundamental software development practices

For those without scripting skills and those new to TestComplete and similar tools the best way to learn is to be gradually introduced by pairing with a more experienced team member and then automating your own tests with code reviews during and on completion of test development. Prior to applying automation in practice it is highly recommended that you read the documentation provided and refer back to it as required throughout the development of a test. Even if on initial reading you don't fully understand all the concepts presented in the documentation your understanding will be clarified with experience and the rate at which you learn from developing tests will be accelerated if you have attempt to understand the documentation first.

The recommended order for reading the documentation is as follows:

1. The Zenith Framework Users' Guide - from the start to the end of this section
2. Learning Scripting with TestComplete and JScript - this is a selection of videos outlines the basics of JavaScript is particularly important to those who have not done scripting before. There is also a TestComplete specific video on debugging covered that make this section worth a read
3. The Zenith Framework Users' Guide - from the following section Framework Structure through to the end of the document
4. API Documentation - although the API Documentation is intended as a reference and not to be read end to end spending some time browsing this documentation will give you valuable insights into what utility functions are available in the framework. The API documentation is accessible from the TestComplete IDE under the topic: "The Zenith Automation Framework"

Core Framework Elements

Framework Structure

Why Structure is Important

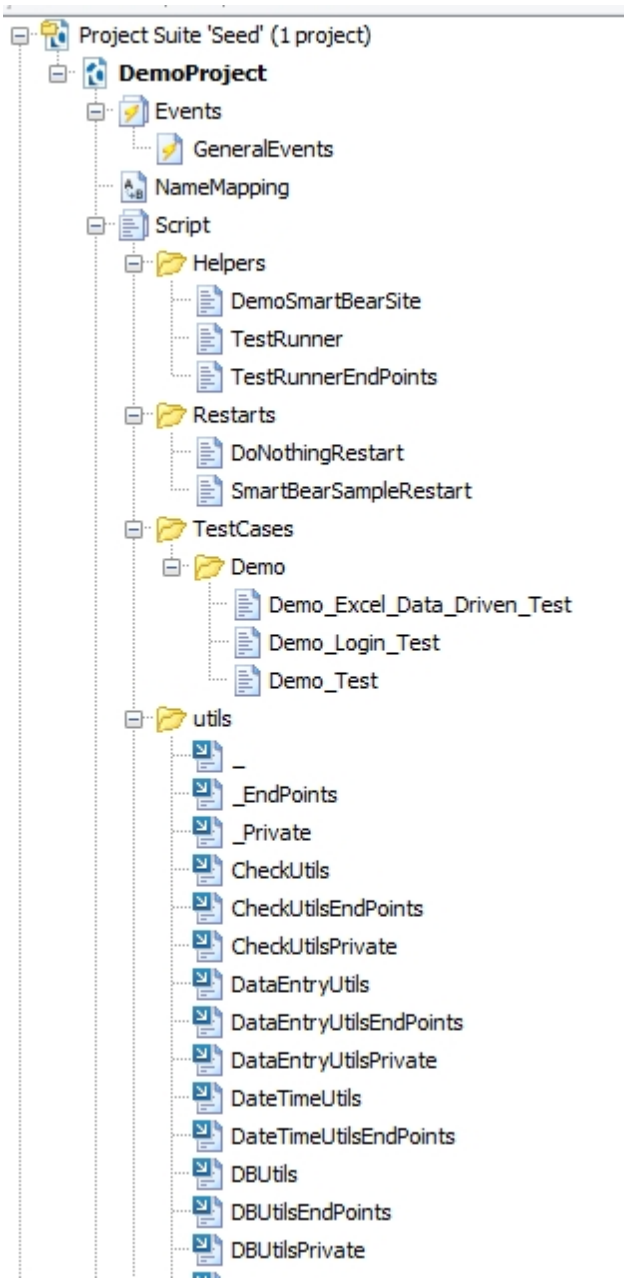
One of the most important principles that needs to be followed if a UI automation project is to be successful is **DRY** - or **Don't Repeat Yourself**. This principle means that any piece of automation functionality should only appear in one place. If two test cases need the same functionality, such as logging in to the application under test, then this logic must be placed in its own function and shared between the two tests. In this way

tests and test functions will mostly be composed of lower level functions rather than being created from scratch and when the application under test changes then the automation suite can be made consistent with the new functionality with minimum effort. This is because the number of places where code needs to be changed is minimised through sharing. Far too frequently automation suites are developed with a disregard for the **DRY** principle and identical logic can be found in hundreds of places throughout the code base. Under such circumstances any non trivial (and sometimes trivial) change can be the death knell of the automation. The effort required to update the automation suite is so great that it is deferred until such a time that there is sufficient resources available to implement the changes required. This time frequently never comes.

What does the framework structure have to do with keeping an automation suite **DRY**?

Before someone implementing an automated test can reuse a function they must know where to look for it. If the framework implements a clear and easily understood structure then the chances of someone looking in the correct place for an existing function is greatly increased.

TestComplete Project Suite Structure



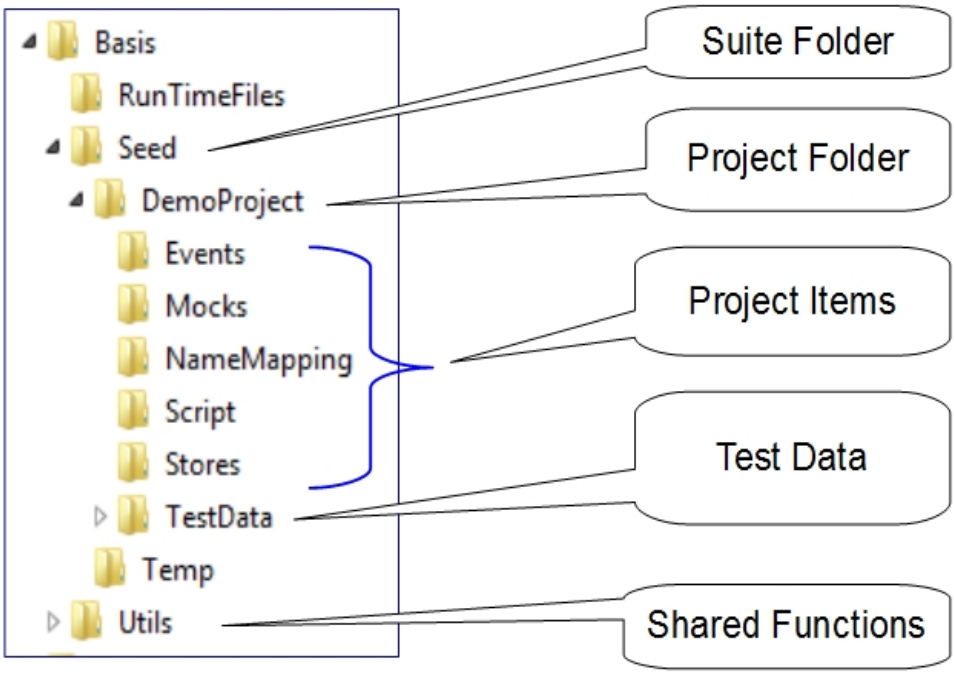
*The above shows a typical TestComplete project suite structure. Every **Project Suite** has one or more **Projects** and every **Project** has a number of different types of **Project Items** such as **Events**, the **NameMapping** object and most importantly **Scripts**. In the example the **Seed** project suite contains a single project **DemoProject**.*

In the Zenith Framework a project suite should correspond to an application under test. So if a company developed a web based booking application and a rich client inventory application you would expect to see two project suites.

In the case of very large automation efforts (hundreds of tests) there may be multiple project suites dedicated to different modules of the same application. The reason for this is that if there are many large projects in a single suite then the IDE (TestComplete) performance can be reduced. It may be slow to open up a test suite and the memory requirements may be high. If a single project gets huge then the syntax check which is executed every time a script is run might also be slow. For a small team it usually takes more than a year to generate enough test cases to make splitting test projects and suites worthwhile. It is

important to note that items such as scripts and the name mapping file can be shared between projects and suites so splitting a project or project suite need not lead to duplicated code.

Directory Structure



The image above shows the directory structure of a typical project suite. In this example there is one project: DemoProject in the suite: Seed. The directory structure more or less mirrors the TestComplete project suite structure with each project in its own sub folder within the project suite and project items contained within the project sub folders. This is the default directory layout TestComplete creates when you create a TestComplete project suite. In addition to this there are a number of special purpose folders that specifically relate to the Zenith Framework. These folders are the following:

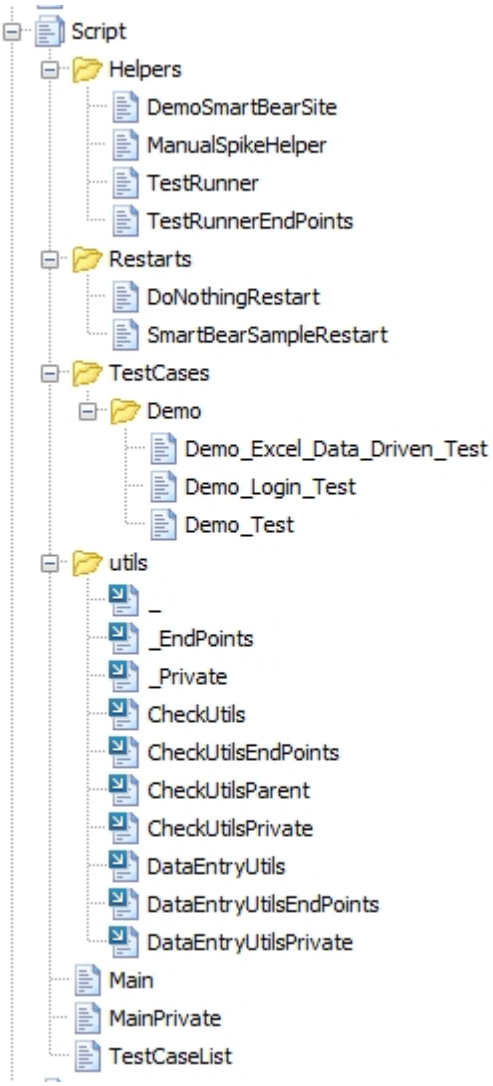
Temp - The Temp directory sits inside the project suite directory and is used by the framework for any temporary files used in running tests. This includes storing the configuration of the last test run or storing data files that need to be manipulated during tests. This directory should not be checked in to version control. If this directory does not exist when needed by the framework then the framework creates it.

TestData - The TestData directory is where test data files such as spreadsheets, text files and database files can be stored. Utility functions that interact with test data files expect the files to exist in this directory by convention. Importantly there is one test data directory per project. The contents of this directory should be added to version control.

Mocks - The Mocks directory is a special test data directory used by the framework to store mock files. Using mock data is an important strategy for reducing the time taken to write and maintain tests. This is discussed in Test Cases and Test Items.

Utils - The Utils directory is a sibling directory to the project suite directory. The Utils directory contains utility scripts that perform generic functions that are shared between multiple projects and project suites. There is more details regarding utility scripts in the sections below.

Script Structure

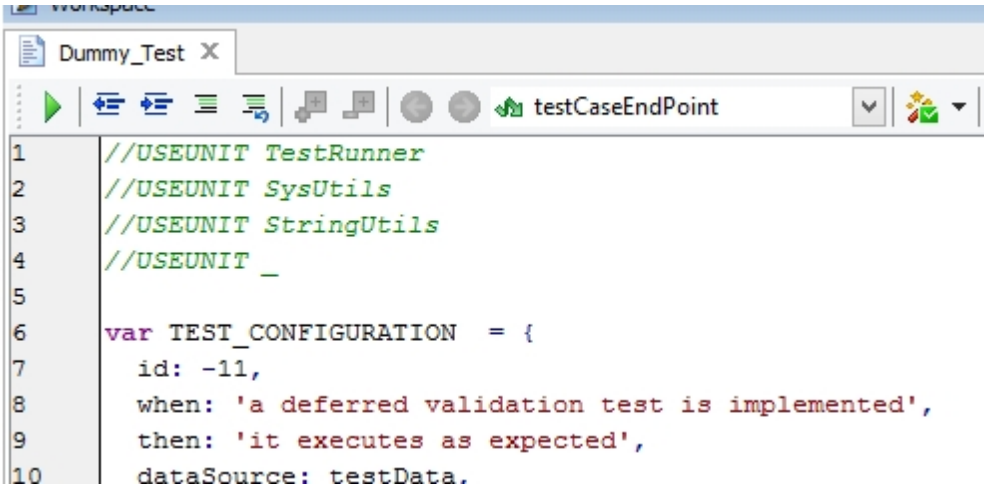


The image above shows how scripts as they displayed under the Script project item in the TestComplete IDE. This illustrates the main types of script within the Zenith Framework. TestComplete does not prescribe any particular script structure or naming conventions but the Zenith Framework does. Each sub-item displayed under the Script project item represents a script file, each of which can contain many functions. The main types of scripts are TestCases, Restarts, Utils, Helpers, the Main script and the TestCaseList. The purpose of these scripts and the relationships between them are discussed in detail below.

Unit Structure

Sharing Functions Between Units

In order for a automation suite to conform to the **DRY** principle, functions must be shared between scripts (also referred to as units) rather than duplicated. A script can then use the functions in another script when a USEUNIT <other script name> clause is added to the top of the script file. The use of USEUNIT clauses is illustrated below.



In the above, the script *Dummy_Test* is able to call functions that are declared in *TestRunner*, *SysUtils* and *StringUtils* and _ by virtue of the three USEUNIT clauses at the top of the file.

Main Unit Types and Dependencies Between Units

The main types of scripts found in the Zenith Framework are as follows:

TestCases - Scripts that execute an end to end test

TestCases are automated test scripts which can be combined into test runs. TestCases start when the application is in its *Home* state, such as the landing page of a web based application, run a test scenario on the AUT and finish with the application back in the same location. There is one test case per TestCase function script file. TestCases have the following characteristics:

- Named <Descriptive Name>+_Test e.g. TimeCost_Salary_Calculation_Test
- Are application (AUT) specific
- Cannot be used (i.e. USEUNIT) by any type of unit other than the TestCaseList. That is a TestCase cannot be used by another TestCase or Helpers or Utils (described below).
- Are discussed in detail in: Test Cases and Test Items

Restarts - *Scripts that execute required set up prior to a batch of tests running*

Restarts reset the AUT to a home start ready to run a batch of tests. They may include code to reset any back end systems such as databases as well as code to navigate to the home state of the AUT:

- Named <Descriptive Name>+Restart e.g. TimeCostRestart
- Are application (AUT) specific
- Can be used by other scripts (but they are mostly used by the test runner)
- Are discussed in detail in Restarts

Helpers - *Scripts that contain shared application or site specific code*

Helpers have the following characteristics:

- Are normally project / application specific
- Contain code shared by two or more test cases
- Are built up as test cases are refactored. I.e. as common functionality is identified in two or more TestCases it is moved into a function in a Helper and the TestCases call this function
- Can use Utils or other Helpers
- Named <Functional Area> e.g. Salary, Tax, Bookings

Utils - *Scripts containing generic functions*

The functionality of Utils scripts forms the basis of the Zenith Framework. Utils have the following characteristics:

- Named <Functional Area><Utils> e.g. StringUtils contains a collection of functions for manipulating strings
- Provide cross project suite functionality and are shared between multiple project suites. No functions specific to an application, such as logging in or navigating to a screen should ever be added to a Utils unit
- The functions in Utils units are often thin convenience wrappers over the top of functions provided by TestComplete e.g. the log function simply calls the TestComplete function Log.Message
- Because functions must remain project independent functions can not use (i.e. USEUNIT) project specific script units such as Helpers or TestCases. Utils can only use other Utils

Main - *A script that defines test runs*

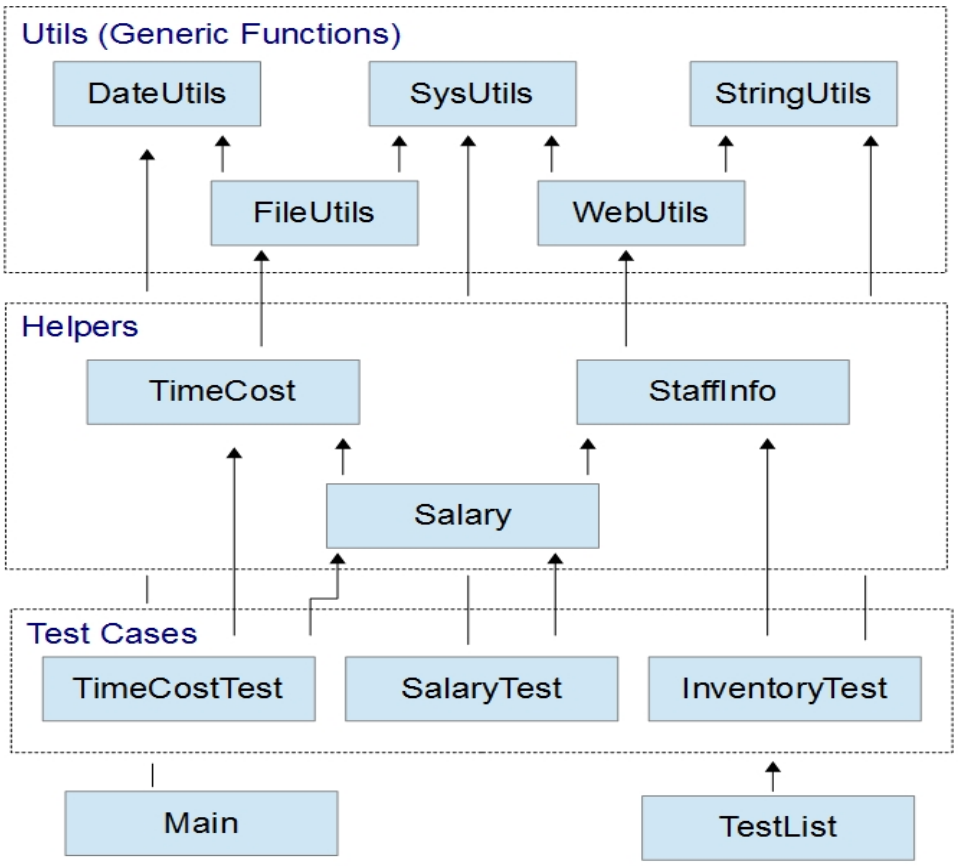
Test runs are controlled by a test runner function. The properties of a test run, such as which tests are to be run, are declared in the run configuration object. There may be many configuration objects depending on how many different types of test runs there are. The test runner function and all test configuration objects are all declared in Main. Main has the following characteristics:

- Main is named - Main 🙂
- Uses what ever generic test runner utils are required to execute a test run
- Is not used by any other script units
- As this is the *composition root* of the framework configuring this is one of the first tasks undertaken by the on-site consultant

TestCaseList - *a list of all the TestCase scripts in a project*.

The TestCaseList is discussed further in Test Runs

Relationship Between Unit Types



The example above illustrates the main unit types and the dependencies that exist between them. Arrows represent dependencies. Because of the clearly defined purpose of each unit type, the naming conventions and the constraints set on their dependencies the resulting unit structure is relatively easy to navigate. When you are looking for a function it should be easy to guess what unit it would be declared in.

Auxiliary Units

In addition to the main unit types there are also auxiliary unit types that improve the usability of framework. A description of these unit types is presented below.

Private - Reduces clutter in the test suite by hiding sub functions used exclusively in one Utility or Helper script.

The public interface of a script such as StringUtils may have a small number of functions intended for use by other scripts. These *public functions*, however, may themselves be composed of many minor sub-functions which are only intended to be used by the script itself (e.g. StringUtils). Putting these *private functions* in their own script reduces clutter when navigating through the actual test code and when using the code completion features within the TestComplete IDE. Private scripts have the following characteristics:

- Are named <Base Script>Private e.g. StringUtilsPrivate
- Are always used exclusively by the base script e.g. StringUtils is the only script which uses StringUtilsPrivate
- Are not necessary for TestCases because TestCases are not used by other scripts

EndPoints - A collection of test functions

EndPoint units are scripts that contain small test functions (EndPoints and UnitTests). The only purpose of these test functions it to invoke and test other functions within the test suite. They do not play a role in testing the AUT. The reason for placing these test functions in a separate script file is to reduce clutter when navigating through the functional test code and when using the code completion features within the TestComplete IDE. EndPoints have the following characteristics:

- Are named <Unit Under Test>EndPoints so test functions for StringUtils would be found in StringUtilsEndPoints and tests for functions in the Salary helper script would be found in SalaryEndPoints
- Are not necessary for TestCases and usually not necessary Private Units. As TestCases are not used by any other script and Private functions are only used by one script there is little to be gained in usability by separating test functions in these scripts into a separate file
- Always use the target script e.g. StringUtilsEndPoints uses StringUtils otherwise it could not access the functions to be targeted by its tests
- Are not used by any other scripts. As theses scripts only contain EndPoints and UnitTests there should be no reason such a script needs to be used by any other
- Details regarding the use EndPoints and UnitTests are discussed in UnitTests and EndPoints

Parent / GrandParent - Used to resolve recursive dependencies in Utils and occasionally low level Helpers.

Interdependencies between two script units is not allowed in JScript. E.g. If StringUtils uses functions in CheckUtils then TestComplete will throw an error if you try to add StringUtils to the Uses list of CheckUtils. The two scripts are not allowed to *use* each other. Creating a script StringUtilsParent and moving the required StringUtils function to that resolves this problem. Parent / GrandParent scripts have the following characteristics:

- Are named <Base Script>Parent or <Base Script>GrandParent
- Functions with the same name and signature that simply call the function in the parent unit (e.g. StringUtilsParent) are placed in the main unit (StringUtils) .

Test Runs

The run<CompanyName>Tests function is configured by the on-sight consultant early in the roll-out of the automation framework. This function takes a single object, that being a runConfig (run configuration) object and calls a framework function (runTests). It is the run<CompanyName>Tests function which invokes the test run.

In MainPrivate - Configured Once by Consultant

```
// This is in MainPrivate script and would be configured
// by the on site consultant
function runMyCompanyTests(runConfig){
    runTests(
      configFileNoDirOrConfigObj,
      defaultRunConfigInfo(),
      defaultTestConfigInfo(),
      preAcceptMethod,
      restartMethod,
      simpleLogProcessingMethod
    );
}
```

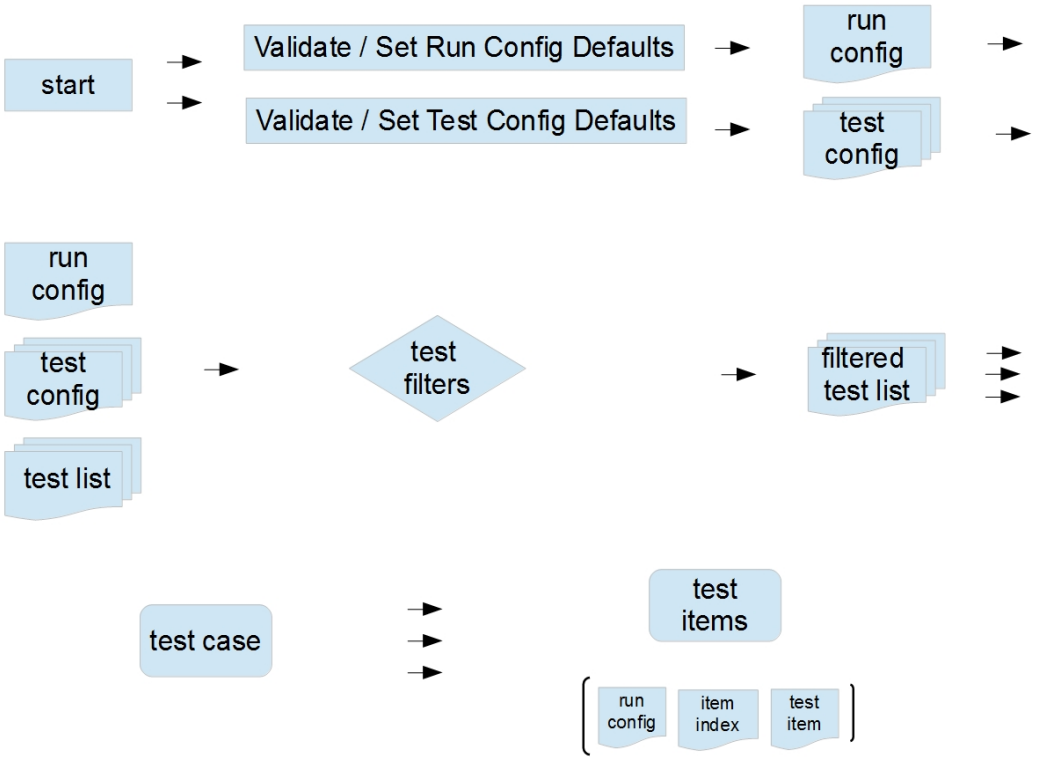
In Main - Configured Once by Users

Different test runs are simply one line functions that call run<CompanyName>Tests using a different runConfig object. RunConfig objects are usually defined within the call to run<CompanyName>Tests (i.e. inline).

```
// functions such as this would be configured by users and added
// to the Main script. It is these functions that trigger test runs
function smokeTestRun(){
  runMyCompanyTests(
                      {
                        name: 'Smoke Test',
                        smokeTest: true,
                        countries: All
                      }
                    );
}
```

Note: There could be dozens of ..TestRun functions in a large test suite

What Happens When a Test Run is Executed?



The image above illustrates the main events that occur during a test run. A test run is executed by invoking a series of function calls which together run a specific subset of a project's test cases under a predefined set of conditions. The information required by these functions, which describe the test conditions, is passed in to the functions via config objects.

There are three basic stages to executing a test run:

1. Validating and, if required, filling in default values in the config objects
2. Filtering the TestCaseList based on the properties of the config objects and the name of the tests
3. Running the test cases in the filtered test list. Many iterations of the same test may be run using the data and validation functions from different test items

Config Objects

Config (configuration) objects are simple JavaScript objects. There are two types of config objects in the Zenith Framework, **runConfig** objects which carry information relating to a whole test run and **testConfig** objects which relate to individual tests. Every test run has one runConfig object passed in from the main test function. Every test case has its own testConfig object declared in a variable called TEST_CONFIGURATION. There are one or two properties on each of these types of objects mandated by the framework, others can be added as per the requirements of the AUT. Config objects will influence how a test and test suite runs and can also be used for logging and reporting. Sample config objects can be seen below:

runConfig

Example

```
// e.g. 1
var smokeTestRunConfig = {
    name: 'Smoke Test',
    smoke: true,
    countries: All
  };

// e.g. 2
var cloudMinimalRunConfig = {
    name: 'Little Cloud',
    tests: ['Rostering*', 'Inventory*'],
    countries: AUSTRALIA(),
    cloud: true
  };
```

Of the properties above, name is the only property required by the framework. Note that the tests property, although not required, has a special meaning in the framework. The tests property can be used to represent the initial test list. All other tests will be excluded.

Another important thing to observe is that not all runConfig properties need to be present on all runConfig objects. In the above example smokeTestRunConfig does not have a cloud property and cloudMinimalRunConfig does not have a smoke property. How the framework deals with these missing properties is discussed further in the sections below.

TEST_CONFIGURATION

Example

```
var TEST_CONFIGURATION = {
  id: 4,
  when: 'a test is driven by an array',
  then: 'it completes as expected',
  owner: 'jw',
  smoke: true,
  enabled: true,
  production: true
}
```

A test configuration (testConfig) object, such as the one above appears in every test case. The id, when and then properties are the only required properties. Other properties are used as required and are specific to the AUT.

Setting and Validating Config Properties

One of the first steps executed by the framework during a test run is to set default properties for both the runConfig object and all the testConfig objects in the project. The optional and mandatory properties for a runConfigs are defined in the `defaultRunConfigInfo` function and there is a similar function which defines these properties for a testConfig called `defaultTestConfigInfo`.

defaultRunConfigInfo

Example

```
function defaultRunConfigInfo(){
  var result = {
      requiredProperties:['name'],
      demo: false,
      smoke: false,
      country: 'Australia',
      // an empty array in the tests property
      // means the tests property will be ignored
      // and all tests in the project could be run
      tests: []
  }
  return result;
}
```

The `defaultRunConfigInfo` function is in the TestRunner helper script and returns an object that is used to validate a runConfiguration. It has one special property called `requiredProperties` that lists all the properties that must be present in a run configuration object. All the other properties (in this example: enabled, demo, smoke, country and tests) are optional and provide defaults to be used when the property is absent. If a run configuration object does not include all required properties or has properties that are not listed anywhere as either required or optional then an error will be thrown at the start of the test run.

The following examples show how this validation would work with the above defaultRunConfigInfo function:

```
// e.g. 1 - INVALID - NO name PROPERTY
var smokeTestRunConfig = {
    smoke: true,
    country: 'Australia'
  };


// e.g. 2 - INVALID - cloud PROPERTY IS NOT DEFINED IN DEFAULT CONFIG INFO
OBJECT
var cloudMinimalRunConfig = {
    name: 'Little Cloud',
    tests: ['RosteringTest', 'InventoryTest'],
    country: 'Australia',
    cloud: true
  };

// e.g. 3 - VALID - the only required property (name) is present and the demo
// property is present as an optional override
var demo = {
    name: 'Run Everything',
    demo: true
  };
```

```
/*
  In the above case the actual object used after the test set up is run would
be like this:
  {
    // a required property
    name: 'Run Everything',
    // optional property overridden
    demo: true,
    // The following default values would be filled in by the framework
    // based on the object returned from the defaultRunConfigInfo function
    smoke: false,
    country: 'Australia',
    tests: []
  };
*/
```

As the requirements of an automation suite evolve and more properties are required to control a test run then this function will need to be updated. Usually by the addition of more optional properties.

defaultTestConfigInfo

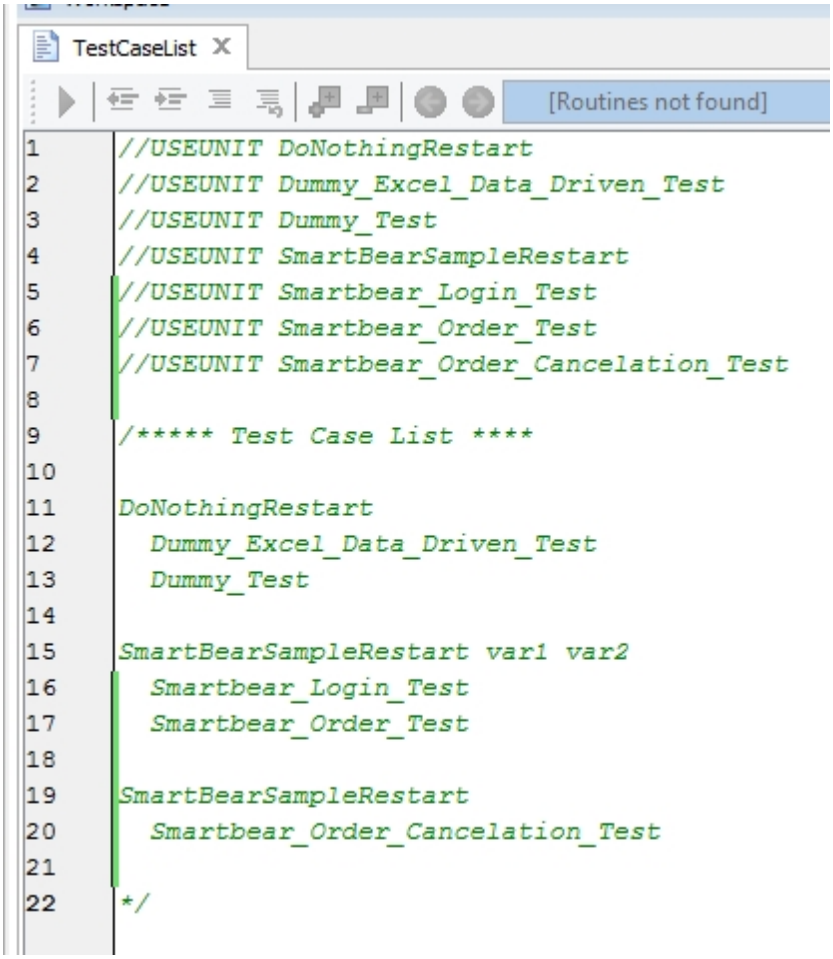
Example

```
function defaultTestConfigInfo(){
  var result = {
    requiredProperties:['id', 'when', 'then'],
    enabled: true,
    demo: false,
    smoke: false,
    countries: ['Australia', 'New Zealand']
  };
  return result;
}
```

The `defaultTestConfigInfo`, like defaultRunConfigInfo function is found in the TestRunner helper script. The object returned by this function is used to validate and fill in defaults on a test configuration in the same way `defaultRunConfigInfo` is used for the runConfiguration. Just as with runConfigs, more properties often need to be added to the result of this function as the requirements of the automation suite evolve.

Understanding the TestCaseList

The Zenith Framework uses a conventions based approach to define the behaviour of test runs that depend on a TestCaseList script file. This approach is easy to maintain, even when a test project starts getting large. The test case also incorporates a simple mechanism for specifying how and when the test suite should restart throughout a test run. Illustrated below is a sample TestCaseList:



The order of the elements in the TestCaseList determines the order in which tests and restarts are executed. The test case list only contains:

1. Unit references (i.e. //USEUNIT <script name>) at the top of the script
2. Restart Script names (e.g. DoNothingRestart) - within the Test Case List comment block
3. Test script names (e.g. SmartBear_Login_Test) - within the Test Case List comment block

Incomplete Test Lists

By convention any script file in a project that with a name ending in the word Test is deemed to be a test case. All test cases in the project should be included in the TestCaseList. When a test run is executed then test list is validated, i.e. TestCaseList is checked to ensure all test cases in the project are present. If the TestCaseList is found to be incomplete and the tests are being run from a batch file (like it would be in an overnight run) then an error is logged. If TestCaseList is being run in interactive (development) mode then an error is logged and the test run aborted. Also at this time, any test ids or restart ids that are null will be assigned a value. The error message will be as follows:

```
Incomplete TestCaseList - the following files are not in the list:...
```

The missing test would be appended to the bottom of the TestcaseList file and the developer would have to move it into the TestCaseList before rerunning.

Filtering Test Cases

Not all tests in a project are execute in every test run. The test list is filtered prior to tests being executed.

The TestCaseList is filtered prior to a test run based on the runConfig, the test script names and their testConfigs. By way of illustration, imagine a test run based on the runConfig and TestCaseList below. Under this configuration you can see the test run is "hard wired" via the `tests` property to only include the following tests: **`Login_Test, Smartbear_Order_Test`**

```
function exampleTestRun(){
  runMyCompanyTests(
                      {
                        name: 'Example',
                        tests: ['Login_Test', 'Smartbear_Order_Test'],
                      }
                    );
}
```

The content of the initial TestCaseList is as follows:

```
DoNothingRestart
   Demo_Excel_Data_Driven_Test
   Demo_Test
   Login_Test

SmartBearSampleRestart var1 var2
  Smartbear_Login_Test
  Smartbear_Order_Test

SmartBearSampleRestart
  Smartbear_Order_Cancelation_Test
```

The TestCaseList would be filtered by the framework as follows:

1. Unwanted test cases are removed

```
DoNothingRestart
  Login_Test

SmartBearSampleRestart var1 var2
  Smartbear_Order_Test

SmartBearSampleRestart
```

In this case, because of the very simple runConfig all this means is that the test cases that match items in the tests property of the runConfig are included and all other tests are excluded.

2. Redundant restarts are removed (any restart that is not followed by a test is redundant)

```
DoNothingRestart
  Login_Test

SmartBearSampleRestart var1 var2
  Smartbear_Order_Test
```

The test run will now execute the following:

```
DoNothingRestart
Login_Test
SmartBearSampleRestart(var1, var2)
Smartbear_Order_Test
```

Test Filters

A set of functions called `testFilters` provide the mechanism by which the framework determines which tests are included in a test run. Default filters are provided with the framework but these may be modified and extended according to the requirements of the AUT.

Example

```
function testFilters(){
  ...
  return [
    function is_enabled(testName, testConfig, runConfig){
      return testConfig.enabled;
    },

    function country_check(testName, testConfig, runConfig){
      return _.contains(forceArray(testConfig.countries), runConfig.country);
    },

    function demo_check(testName, testConfig, runConfig){
      return testConfig.demo === runConfig.demo;
    },

    function is_in_test_list(testName, testConfig, runConfig){
      return nameInList(testName, runConfig);
    }
  ];
}
```

Prior to any test being run each function returned from `testFilters` is run against each test is the test list. For a test to be included in a test run every test filter function must return true. A test filter function is simply a descriptively named function that takes the `testName, testConfig and runConfig` as parameters and returns true or false.

After the test list is filtered each of the test that remains in the test list and any associated restarts are then run in the order they appear in the list. Different companies will create different test filters based on their requirements.

Restarts

Restarts control how the test environment is reset and the AUT is started and set to its "home state" prior to each test. Restarts are discussed in detail in Restarts

Test Cases

If a test case is not filtered out by the test filters then the test case will become part of the test run. A single test case will usually be repeated multiple times (iterations) in one test run based on different data (test items).

The details of test cases are discussed in the next section: Test Cases and Test Items

Test Cases and Test Items

As outlined in Test Runs a test case is a single test script that encodes a test case definition which can be repeated one or many times in a test run with different data and validation logic. The objects that carry the data and validation logic for one execution (or iteration) of a test are called "test items". For example a login test case might have separate test items to verify:

- a user can login with valid credentials
- a user gets the appropriate error message when their user name is incorrect

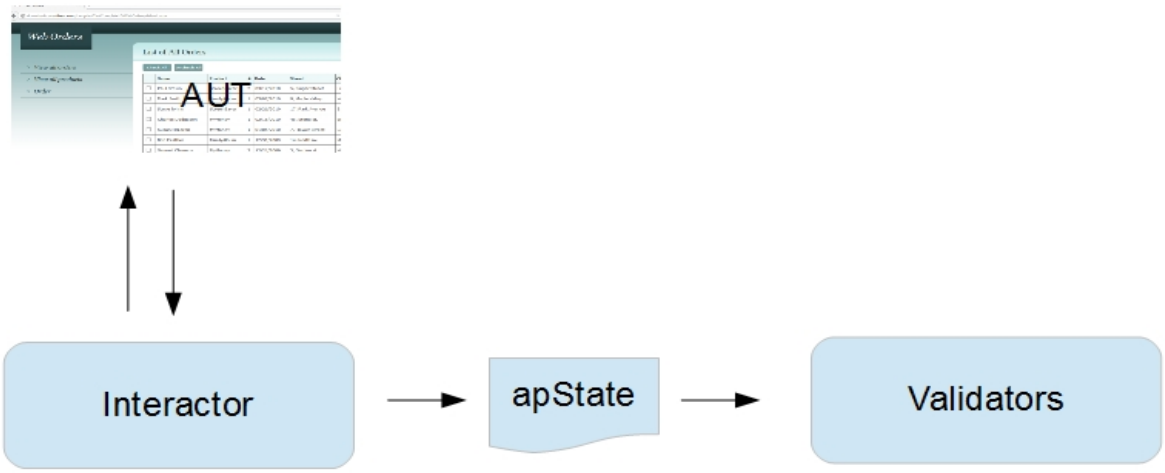
A test script must include a test case definition and function that returns a list of test items (called `testItems`). During a test run one iteration of a test script is executed with each element returned by the `testItems` function.

```
// Test Execution and Validation Code Omitted

function testItems(runConfig){
  return [
    {
      id: 1,
      when: 'the correct login credentials are used',
      then: 'the user is able to log in',
      userName: 'Tester',
      password: 'test',
      validators: logged_in_successsfully
    },
    {
      id: 2,
      when: 'the user name is incorrect',
      then: 'the user is not able to log in and gets the expected error
message',
      userName: 'Tester1',
      password: 'test',
      validators: [
                    log_in_should_have_failed,
                    error_message_as_expected
                  ]
```

```
        }
    ]
}
```

The above is an example illustrates how data is provided to a test via a `testItems` function. Every test case must have a `testItems` function that will return an array with of at least *one record. In the example above the test would be run twice (two iterations), once logging in with the correct test credentials and once attempting a login with an incorrect user name.*

The Two Stages of Test Execution



The execution of a single iteration of a test case occurs in two stages, an interactor stage followed by a validators stage. In the interactor the test script interacts with the AUT and records information about the state of the application as relevant to the business rules that are subject of the test. The object to which the reader appends this information is called the `apState` object. An example of the kind of information a reader might record would be, in a login test, the URL the user is redirected after they click the login button and the text of any error messages visible.

When the interactor has finished then the `apState` object is passed on to the next stage of execution, validators. In the this stage any functions (there may be 0 to many) are assigned to the validators property of the current test item are executed against the `stateInfo` object and success or failure is logged. Returning to the login example, in the case where login success is expected then the attached validator would check the URL after login is the application home page. In login failure scenarios validation would check the URL after log is still the login page and separate validator would check for the appropriate error message text.

The Naming Conventions and Structure of a Test Case

As briefly described in Framework Structure, test cases in the Zenith Framework are simply JavaScript test scripts that conform to the framework's naming conventions and structural requirements.

1. The naming convention for test script is that the script name ends with _Test.
2. The structural requirements for a test script (or case) are as follows:
    - the script must not be referenced by any other script (i.e. shared functions or data cannot be stored in a test case)
    - the script must contain all the elements (functions and objects) required by the framework to execute both stages of the test, both during development and within a test run. There are also optional elements which may be implemented if required.

The Elements of Test Case

The following is an example of a full test case script. Each part of this script is discussed separately below.

```
//USEUNIT TestRunner
//USEUNIT SysUtils
//USEUNIT StringUtils
//USEUNIT FileUtils
//USEUNIT _
//USEUNIT DemoSmartBearSite
//USEUNIT WebUtils

var TEST_CONFIGURATION  = {
  id: -2,
  when: 'a user attempts to log in',
  then: 'access is granted or denied as expected',
  owner: 'LLCJ',
  demo: true,
  enabled: true
};

function testCaseEndPoint(){
  var testParams = {
                     testConfig: TEST_CONFIGURATION,
                     itemSelector: all,
                     mocking: false,
                     demo: true
                   };
  runTestCaseEndPoint(testParams);
}

function interactor(runConfig, item, apState){
  logInSmartBear(
                  item.userName,
                  item.password
                );
  apState.url = activeUrl();
  apState.details.someProrperty = 'a really big string';
  var statusLabel = seekByIdStr(0, 'ctl00_MainContent_status');
  apState['error message'] = hasValue(statusLabel) ? statusLabel.contentText:
'N/A';
}

function mockFileNameNoExtension(item, runConfig){
  // use default naming convention
  return null;
}

function logged_in_successsfully(apState, item, runConfig){
  checkEqual(WEB_ORDERS_DEFAULT_URL(), apState.url);
}

function log_in_should_have_failed(apState, item, runConfig){
  checkContains(apState.url, 'Login.aspx');
}

function error_message_as_expected(apState, item, runConfig){
```

```
  checkEqual('Invalid Login or Password.', apState['error message']);
}

function testItems(runConfig){
  return [
    {
      id: 1,
      when: 'the correct login credentials are used',
      then: 'the user is able to log in',
      userName: 'Tester',
      password: 'test',
      details: {
            property: 3
      }
      validators: logged_in_successsfully
    },
    {
      id: 2,
      when: 'the user name is incorrect',
      then: 'the user is not able to log in and gets the expected error
message',
      userName: 'Tester1',
      password: 'test',
      validators: [
                    log_in_should_have_failed,
                    error_message_as_expected
                  ]
    }
  ]
}

function testItemsEndPoint() {
  var runConfig = {
                  };
  var result = testItems(runConfig);
  toTempReadable(result);
}

;(function register(){
  registerTestRunElement(TEST_CONFIGURATION);
}())
```

Note: A template for test case scripts is available in the code templates installed as part of the Zenith Framework

The elements (JavaScript objects and functions) of a test case are as follows:

A TEST_CONFIGURATION Object (Required)

```
var TEST_CONFIGURATION  = {
  id: -2,
  when: 'a user attempts to log in',
  then: 'access is granted or denied as expected',
  owner: '',
  demo: true,
  enabled: true
};
```

The TEST_CONFIGURATION object has properties that are required by the framework for filtering tests and logging test information within a test run.
For example, the enabled property is used by test filters prior to a test run and the when / then properties are used to generate the title for the test in the test log. More information about how configuration objects are used can be found in Test Runs.

A testCaseEndPoint Function (Required)

```
function testCaseEndPoint(){
  var testParams = {
                    testConfig: TEST_CONFIGURATION,
                    itemSelector: all,
                    mocking: false,
                    demo: true
                   };
  runTestCaseEndPoint(testParams);
}
```

The `testCaseEndPoint` is a parameterless function and is only used in development. The purpose of `testCaseEndPoint` is to enable a single (or all) the iterations of a test case to be launched and debugged without needing to set up a test run. Using TestCaseEndPoints is such a crucial part of test development that there is a whole section dedicated to their use: TestCaseEndPoints - A Special Kind of EndPoint.

An interactor Function (Required)

```
function interactor(runConfig, item, apState){
  logInSmartBear(
                  item.userName,
                  item.password
                );
  apState.url = activeUrl();
  apState.someProrperty = 'string info...';
  var statusLabel = seekByIdStr(0, 'ctl00_MainContent_status');
  apState['error message'] = hasValue(statusLabel) ? statusLabel.contentText:
'N/A';
}
```

The `interactor` function is invoked by the test runner in order to complete the first stage of a test iteration. This is interaction with the AUT and recording of application state information as described above. When the interactor is executed it is passed the following parameters:

- **`runConfig`**: properties relating to the test run (e.g. whether the test run is a full or smoke test run or if the test run is targeting a specific country). This information can be used to modify how the test executes depending on the test run. E.g. a test for online accounting package may run

differently based on different tax rules for different countries

- **`item:`** this is the test data to be used for the current test repetition i.e. it corresponds to a single record in the `testItems` list. The properties of this object are almost entirely dependent on the test case and can include data such as input values and functions such as function that navigates to a particular screen
- **`apState:`** this is the object to which the interactor adds information about the state of the AUT. At the start of a test iteration (there is one iteration executed for each test item) the `apState` object will be empty but for a single `details` property which is itself an empty object. The purpose of the reader is to populate the `apState` with information relevant to the business rules under test. Supporting data (such as the full text of a REST service response) may also be added to the apState. This information can be useful when doing a detailed analysis of a test failure but is not essential to understanding the intent of the test. The `apState` object is validated in the next stage of the test.

Validator Functions (Required for all tests which validate business rules, which should be almost all tests)

```
function logged_in_successsfully(apState, item, runConfig){
  checkEqual(WEB_ORDERS_DEFAULT_URL(), apState.url);
}

function log_in_should_have_failed(apState, item, runConfig){
  checkContains(apState.url, 'Login.aspx');
}

function error_message_as_expected(apState, item, runConfig){
  checkEqual('Invalid Login or Password.', apState['error message']);
}}
```

After the interactor is complete the `apState` object that has been built by interactor is passed on to any validator functions attached to the test item to verify the business rules that are the target of the test case. These functions take three parameters, `apState`, `item`, `runConfig` (as described above for the `interactor` function). Validator functions are attached to the `validators` property of a test item object, so each iteration of a test case can apply different validation rules.

There can be zero, one or more validator functions assigned to the `validators` property of a test item. If there is a single validator function then this function can be assigned directly to the `validators` property on the test item, if there are multiple validator functions then an array of functions is assigned to the `validators` property of this item. If there no validator then the only errors in the test log will be due to the reader failing (e.g. when the reader cannot find an expected element in a web page).

By convention validator functions are named in snake case and should be clear and descriptive e.g. `user_logged_in_successfully(stateInfo, params, runConfig)`. This makes the test log summaries easy to read should any of these validations fail.

A testItems Function (Required)

```
function testItems(runConfig){
  return [
    {
      id: 1,
      when: 'the correct login credentials are used',
      then: 'the user is able to log in',
      userName: 'Tester',
      password: 'test',
      details: {
                bigRequestString: 'a really big string..'
               }
      validators: logged_in_successsfully
    },
    {
      id: 2,
      when: 'the user name is incorrect',
      then: 'the user is not able to log in and gets the expected error
message',
      userName: 'Tester1',
      password: 'test',
      validators: [
                    log_in_should_have_failed,
                    error_message_as_expected
                  ]
    }
  ]
}
```

The `testItems` function is the basis for data driven testing in the Zenith Framework. This is a JavaScript function that returns an array of objects, each of which form the test data and validation functions for a single test iteration. These objects are referred to as test items. The `testItems` function takes a single parameter, runConfig. The runConfig parameter can be useful for filtering the array returned based on properties of the test run. For example, if the test run was a smoke test (the smoke property on the runConfig was true) then all but a few test item objects would be filtered out of the array (i.e. only a few iterations of the test would be run).

The properties of the individual test data objects will be almost entirely dependent on the test case. A unique (within the test case) id property is, however, required for every object and it is best practice to include when / then properties to enable the framework to produce informative logs at runtime. The id property makes it easy to identify and run a single item for development and debugging. All test cases require a `testItems` function even singular test cases. If a test case has only one item then the array returned by the `testItems` function would only contain one object.

It is possible, but not recommended, to generate the array of objects returned by the `testItems` function from a spreadsheet. This topic is discussed in: Data Driven Testing Using Excel.

A testItemsEndPoint Function (Useful)

```
function testItemsEndPoint() {
  var runConfig = {
                  };
  var result = testItems(runConfig);
  toTempReadable(result);
}
```

The `testItemsEndPoint` is simply a function that enables the user to invoke and check the result of

testItems function during development. See Unit Tests and EndPoints for details on how these types of functions are used.

A mockFileNameNoExtension Function (Optional - Implemented Rarely)

```
function mockFileNameNoExtension(item, runConfig){
  // use default naming convention
  return null;
}
```

An important feature of the Zenith Framework is ability to save the apState object from a test run or fake this data then develop business rule validation independently of the AUT. This saved or faked data is called mock data, see UnitTests and EndPoints for details on using mock data. Mock data is stored in text files. By default the framework names these using a combination of the script name and the item id. In rare cases this may not be suitable, for example if you want to use different mock data for test runs of different countries so you would need the function to produce different names based on the properties of the runConfig as well as the item id. For this reason a `mockFileNameNoExtension` function can be included in a test case script. In the vast majority of cases there is no need to provide a `mockFileNameNoExtension` function.

A Registration Block (Required)

```
;(function register(){
  registerTestRunElement(TEST_CONFIGURATION);
}())
```

The registration block sits at the bottom of the test script is required by the framework so tests a registered at start up. You should never delete or modify this function.

The Advantages of Two Stage Test Cases

Executing test cases in two phases as in the Zenith Framework is an implementation of a very common approach in software development. That is to isolate complexity and/or error prone or slow processes. In this kind of automation test the interaction with the AUT is usually the most complex part of writing an automation script and waiting for this part of test to to run can also be quite a time consuming. Using a two stage process enables the test developer to:

1. Write tests more quickly by pre-recording application state and developing business rule validation against this pre-recorded state rather than waiting on the AUT for test case development
2. Write and test business rule validation before an application is developed

These topics are covered in more detail in Test Development Workflow

There are also advantages in using a two stage approach when reviewing the results of test runs:

1. There is a more immediate indication of the cause of errors when errors are logged.

   Errors logged during the interactor stage of test will always be due to:
   i. bugs in the test case or
   ii. the AUT changing in such a way that the automation is no longer consistent with the behaviour of the application or
   iii. a major defect in the application that is not targeted by the test case (e.g. a null pointer exception)

   In contrast, errors logged during the second (validation) stage of a test run will almost always be application errors in the business rules targeted by the test.

2. When business rule errors are detected then the test summary provided in the test log gives a very clear indication of the nature of the error and can form most of a log in a defect tracking system. The test summary includes:
   i. the test item data
   ii. the apState object which summarises the state of the application throughout the test
   iii. the name of validation rules run and the text of any error messages logged when these rules were run

   This is just the kind of information a developer needs when they are trying to understand and reproduce a defect.

Restarts

For most tests, assumptions need to be made about the initial state of the application's back end and front end for a test to be executed successfully. For example, a test relating to how an application performs when an inventory item is deleted requires the target item to be in the database at the beginning of the test. It may not be possible to repeat such a test without resetting the database or manually reinserting the item. Also user facing applications, such as web applications, tend to have an intuitive "home state" (such as logged in and on the user home page of a web application) from which a test would start.

In the Zenith Framework the initialisation of the AUT's back end and navigation to the application's home state is implemented in specialised scripts known a "restart scripts". The functions contained in these scripts are executed as required by the framework's test runner such that each test starts from with the application in its home state and the application's back end is (optionally) refreshed before every batch of tests.

Restart scripts have the following characteristics:

1. Are named ending in the word Restart
2. Contain all the functions required for the framework to manage starting and restarting of test

cases. These are:

- `rollOver` - resets the test environment back end. May include code resetting databases, VMs, containers or devices. The implementation of this function can be left empty if no such functionality is required by the test batch
- `goHome` - opens the application if it is not already open and navigates to its home state. E.g. opening a browser and logging in or clicking on the <Home> button if the user is already logged in
- `isHome` - a function that returns true if the application is in its home state
- `close` - closes the application

3. Are included in the TestCaseList as described in Test Runs

How Restarts are Implemented

Listed below is an example restart script called `SalesRestart`. The testing and framework registration code has been removed

```
function rollOver(runConfig, paramStr){
  resetDatabaseSnapshot(runConfig);
}

function goHome(runConfig, paramStr){
  var link = seekInPage({ObjectType: 'Link', contentText: 'View Orders}, 0);
  if (link.Exists){
    link.Click();
  }
  else {
    logInSalesSite();
  }
}

function isHome(runConfig, paramStr){
  return hasText(activeUrl(), '/acmeSales/ordersView.aspx');
}

function close(runConfig, paramStr){
  closeBrowser();
}
```

The implementation of these functions is quite straight forward:

`rollOver:` Calls a helper function which restores the test database from a snapshot

`goHome:` Checks to see if there is a <view orders> link visible and clicks on this link if it is. If the link is not visible, then a helper function is called to log in to the sales site

`isHome:` Returns true is the user is already at the view orders screen

`close:` Closes the browser

A restart script template which includes an empty implementation of the above functions is included in the Zenith Frameworks default code templates.

How Restart Functions are Called in a Test Run

The position of restart scripts in the TestCaseList is what determines what restarts are applied to what tests. Put simply: The active restart script for any batch of tests is the restart script listed at the top of the batch, this script's `rollOver` function is called prior to the batch running. Between any two test iterations `isHome` is called to determine if the AUT is in its home state and `goHome` is called if it is not.

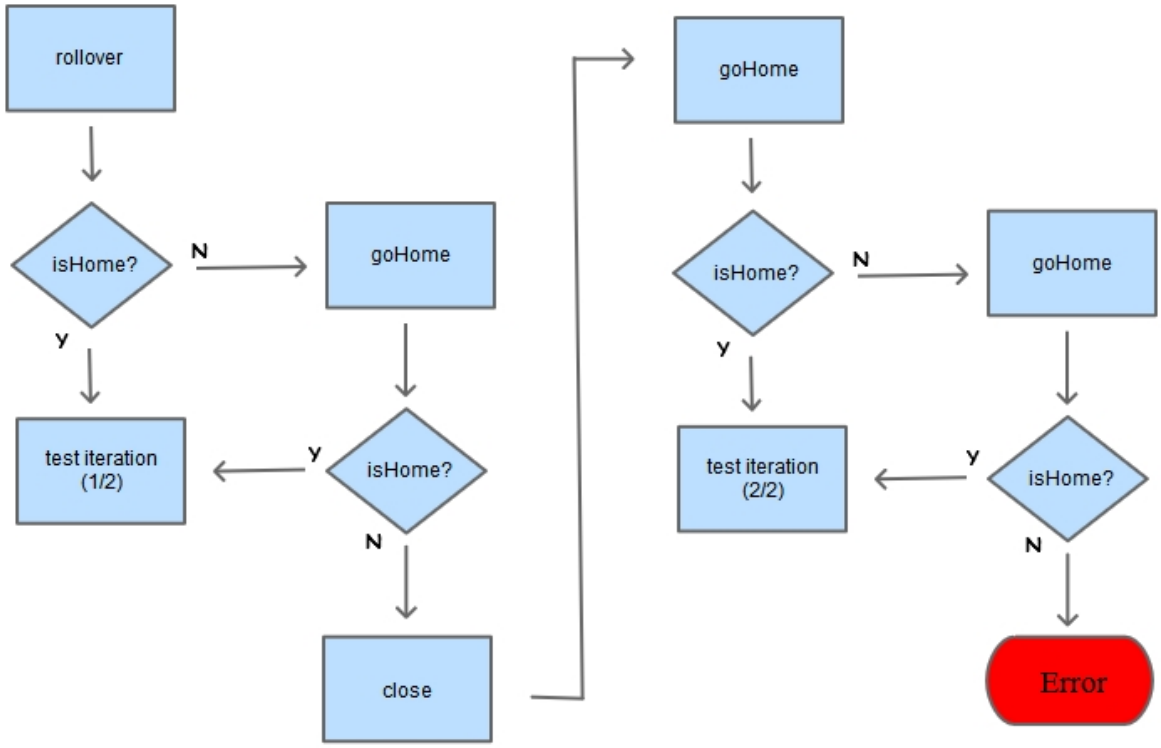
```
SalesRestart
  Sales_Enquiry_Test
  Sales_Order_Test

SalesRestart
  Order_Cancelation_Test
```

In the above test list the `SalesRestart rollOver` would be executed prior to the first script in the following batch. This would be the *`Sales_Enquiry_Test`* if it has not been filtered out prior to the test run commencing. This restart script would also ensure that the AUT was in its home state before each iteration of any of the tests in the list.

In the example above it would be important that the *`Sales_Enquiry_Test`* did not change any data in such a way that would interfere with the *`Sales_Order_Test`* because they are in the same batch meaning there is no `rollOver` between the two.

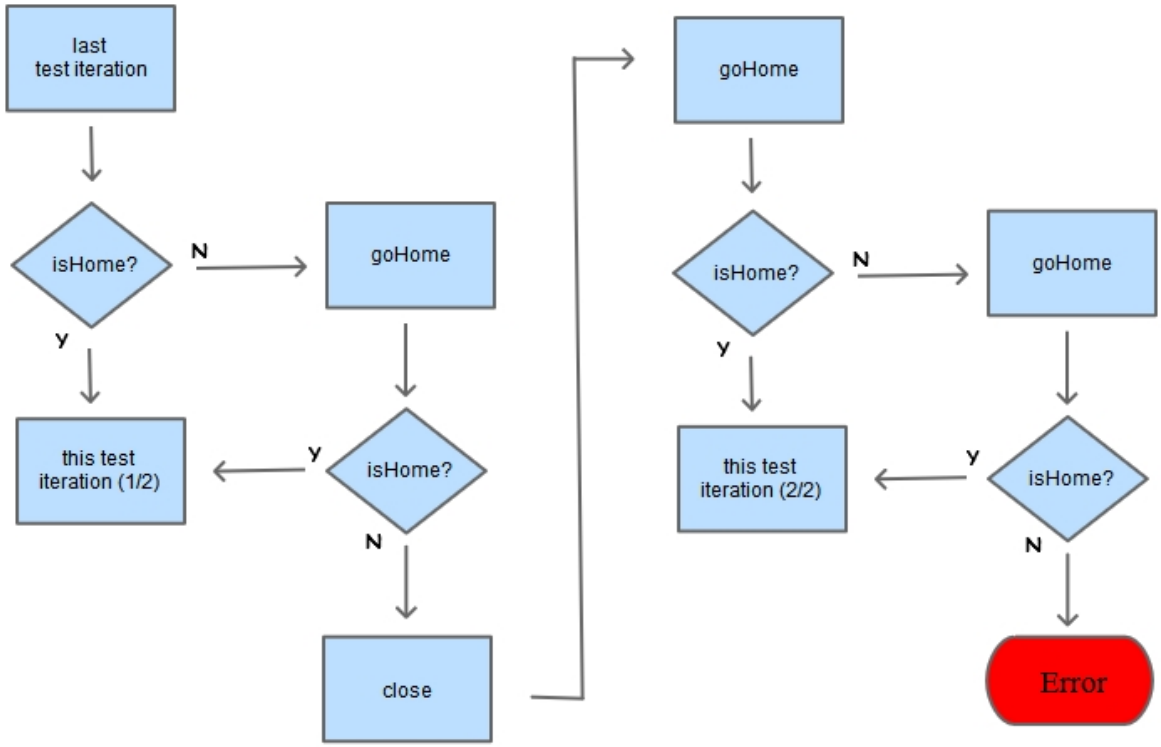
Below are some more detailed flowcharts of how Restarts interact with tests.

Rollover and Navigation at the Start of a Test Batch



This is the same control flow as described above but the test runner also includes a retry when navigation to the application home state if the initial attempt is not successful

Navigation Between Test Iterations



Navigation between test iterations is identical to the start of a test batch except that there is no RollOver

The Advantages of Specialised Restart Scripts

Having specialised restart scripts that are implemented and invoked in a consistent way allows the test developer focus purely on the functionality under test rather than dealing with application initialisation within each test case. Once the user becomes familiar with the control flow described above, the test suites developed are much more robust and maintainable than if AUT and test environment initialisation was implemented in a piecemeal test by test fashion.

Using Zenith

The purpose of this section is outline some of the most important conventions and features of the Zenith Framework. It is not intended to be a detailed How To or Step By Step guide to the framework's use. Other Resources are available to assist with a more detailed understanding of the functions provided by the framework, TestComplete and JavaScript. There is, however, no substitute for experience so it is vital to get as much hands on use of the framework while the consultant is on site.

Test Development Workflow

Automation Planning

Prior to commencing automation it is important to gain an overview of the functionality of the application. This can be done by reviewing users guides, interviewing business or IT staff, or reviewing existing manual test cases. Regardless of how this information is gathered a clear understanding of the AUT is essential to devising the overall automation strategy.

The details of what is tested should be based on the likelihood of a defect occurring, the costs associated with a defect in a particular area and also, through discussions with the development team, the anticipated level of code churn in the different areas of the application. By understanding these factors it will then be possible to develop a common sense list of automation candidates.

Generally it is recommended to plan to automate the AUT broadly rather than narrowly and deeply. As the automation suit develops it will then be possible to write more detail tests for the more important or risky areas of the application. The selection of automation tests should also be weighted towards test scenarios that are heavily data dependent and as such are repetitive and time consuming to execute manually.

Once an outline of a test strategy is created it is possible to create the names of the candidate tests based on their functional area. The full test list should then consist of logically name spaced test cases. For example a back office business application may include tests such as:

```
Finance_Accounts_Receivable_Test
Finance_Accounts_Payable_Test
Finance_Reports_End_of_Month_Test
Finance_Reports_End_of_Year_Test
Payroll_Entitlement_Calculation_Test
Payroll_Leave_Tracking_Test
```

Although the above is a simplistic example (real test cases would be much finer grained than those listed) it does illustrate the name spacing style naming conventions used in Zenith. Adhering to such a convention makes it relatively easy to find test cases and gain an overview of test coverage as the automation suite gets larger.

How Granular Should a Test Case Be?

All Zenith test cases are data driven and the recommended means of providing this data is via an array of JavaScript objects. This is particularly powerful as functions can be passed to a test case as object

properties and these functions can be used change the way the test case interacts with the AUT. This flexibility, however, should not be overused because this can result in different iterations within the same test case interacting with the AUT very different ways. When individual test iterations vary greatly from each other it becomes unclear to other users what functionality is covered by the test and also becomes more difficult to maintain.

If a test becomes difficult to name or write a sensible test case level when / then properties for, or if there are a lot of test item function properties that change the test's interaction with the AUT, then this may be an indication that the test script needs to be split into two or more separate more focussed scripts.

If existing manual test cases are to be used as a basis for writing automated tests then it is usually necessary to break these down into more focussed test cases as well. Because manual tests are slow and expensive to run, manual test cases will often be more like tours. These tests require the tester to navigate through a long workflow and verify a lot of application state along the way. In contrast to manual tests, automated tests are relatively quick and inexpensive to run but more difficult to debug and reconcile when errors are encountered. This being the case, a greater number of short concise test cases is more appropriate for automated tests because then when errors occur they are quicker to reproduce and easier to fix.

Adding a Test Case

Adding a test case in the Zenith Framework is as simple as:
1. Adding a test script file to the project that conforms to the naming conversions discussed above
2. Filling in the file with the contents of the testCase code template (code templates can be accessed with <CTRL><J> inside the script editor)
3. Running the testCaseEndPoint in the new script (this function is discussed in detail in TestCaseEndPoints - A Special Kind of EndPoint).
   This will cause an id to be assigned to the test case and the `TestCaseList` to be updated as described in Test Runs
4. Update the TEST_CONFIGURATION object at the top of the test case to accurately describe the intent of the test case

```
var TEST_CONFIGURATION  = {
  id: 5,
  when: 'an inventory item is added or removed',
  then: 'stock available and on hand calculations are correctly executed' ,
  owner: 'JM',
  enabled: true
};
```

Example of a completed TEST_CONFIGURATION object

At this point test case development is ready to commence.

The Two Stages of Automating a Test Case

The two stages of automating a test are:

1. Developing the interactor which interacts with and records the state of the AUT (see Test Cases and Test Items for more details)
2. Developing the validator functions for each test item which validate business rules (see Test Cases and Test Items for more details)

The interactor Function

The `interactor` (function) is implemented by composing calls to sub-functions which may be existing functions from helper scripts or may be written from scratch and placed in the test case script file itself. Any new functions should be developed and debugged in isolation by creating UnitTests or EndPoints (described in detail in the next section) for each function. When the interactor is complete it should consist entirely of calls to a small number of high level functions which can be read like a high level manual test. When the interactor has finished running the apState should have been updated with all the information required to perform business rule validation.

```
function interactor(runConfig, item, apState){
  logOnWebOrders(item.userName, item.password);
  apState.url = activePageUrl();
}
```

In the above login test example, all that is required for validation is a record active url after login. The preceding interaction with the AUT is made clear from the name of function: `logOnWebOrders`

Validator Functions

Once the `apState` is recorded the validator functions can be developed to verify the business rules that are the target of the test.

```
function logged_in_successsfully(apState, item, runConfig){
  checkEqual(WEB_ORDERS_DEFAULT_URL(), apState.url);
}

...

function testItems(runConfig){
  return [
    {
      id: 1,
      when: 'the correct login credentials are used',
      then: 'the user is able to log in',
      userName: 'Tester',
      password: 'test',
      validators: logged_in_successsfully
    }, ...
```

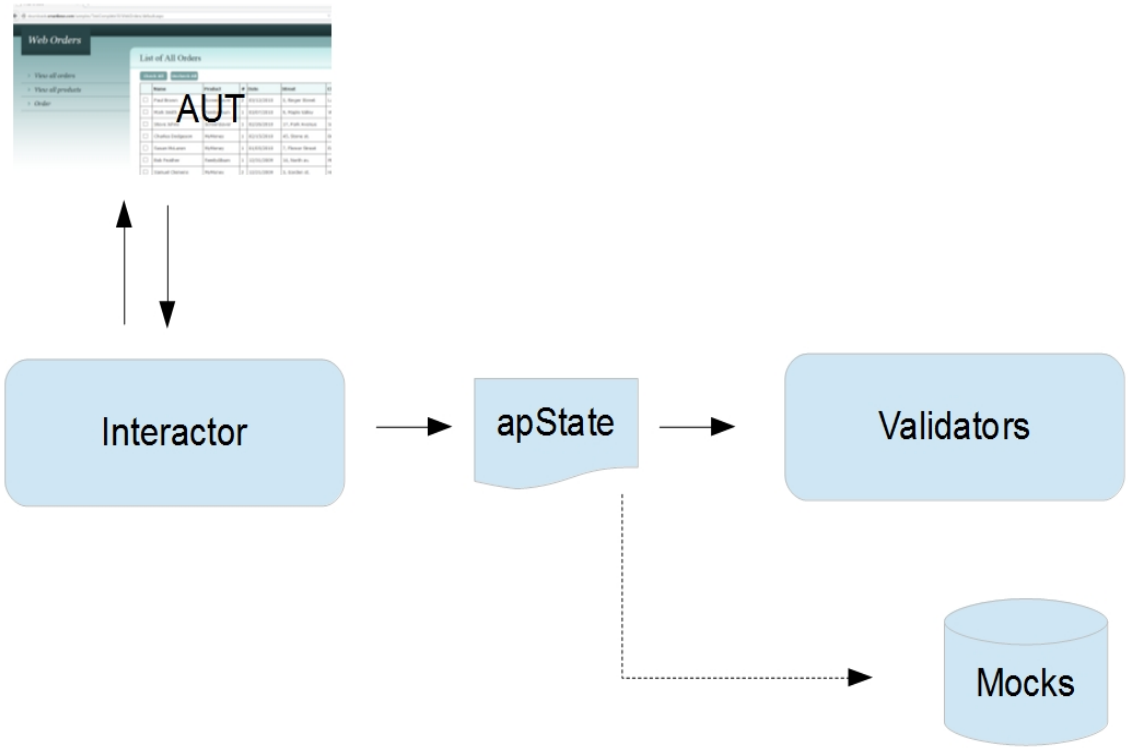
The above demonstrates the a typical validator function (`logged_in_successsfully`*) and how it is associated with a test item via the* validators *property*

Development Workflows

Depending on the nature of the test case the development of the interactor and validators can be performed in different orders resulting in different development work flows. The three main workflows are described below:

Standard Workflow

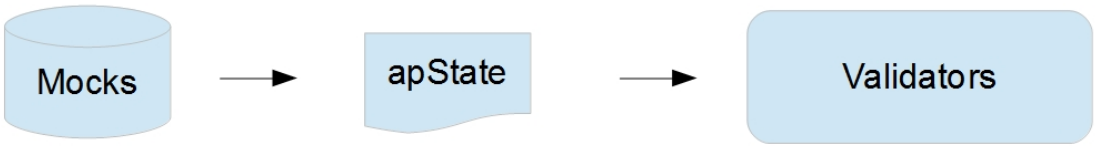
For test cases where the AUT is deployed and ready for test and the number of test iterations is small, test cases can be developed in the same order they are executed during a test run. The interactor is developed and run then validator functions are developed based on the `apState` object built by the interactor. As the `apState` is saved for each test iteration, it is also possible to run the interactor once and develop the validators against the mock data saved when the interactor was run. The details of how to use mock data are discussed in: TestCaseEndPoints - A Special Kind of EndPoint



Standard development workflow follows the same path as test execution

Interactor First Workflow

For test cases with large numbers of iterations it is usually less time consuming to create the interactor and run all test iterations on a separate machine. The mock files can then be checked in to version control and checked out on the development machine and all validator functions can be developed against the mock files.

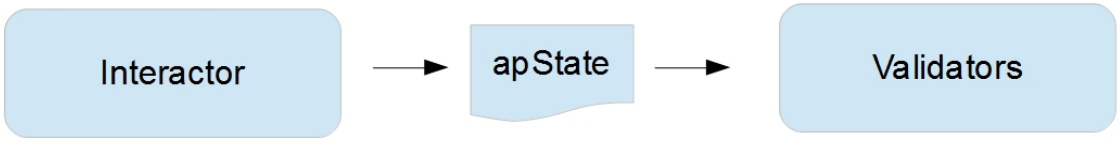


Developing validator functions against pre created mocks can save a lot of time when a test case has many iterations

Interactor Last Workflow

If the AUT is not deployed it is still possible to develop business rule validation (validator functions) by declaring the expected `apState` on the test items and have the interactor simply copy this to the runtime `apState`. In such cases the `interactor` is not actually interacting with the AUT at all in this phase of development.

When the AUT is deployed the "copy through" code can be replaced with genuine AUT interaction code, the mock `apState` can be removed from the test items and tests rerun against the AUT to confirm they are correct.



If the AUT is not deployed then the interactor *can initially build the* `apState` *without the AUT*

```
function interactor(runConfig, item, apState){
  _.defaults(apState, item.apState);
}

function testItems(runConfig){
  return [
    {
      id: 1,
      when: 'the correct login credentials are used',
      then: 'the user is able to log in',
      userName: 'Tester',
      password: 'test',
      apState: {
                 url: WEB_ORDERS_DEFAULT_URL(),
                 'error message': 'N/A'
               }
    },
```

The above demonstrates how the apState is copied from a testItem so validators can be developed in the absence of the AUT

UnitTests and EndPoints

It is highly unproductive to attempt to develop and debug test scripts by running the test script as part of a test suite or even invoking a whole test. An automated UI test could take as long as ten minutes to run a single iteration. If you had to run through the code six times to fix a bug at the end of such a test this would take an hour. The solution is to break down a test case into a sequence calls to small functions and develop each function in isolation. How could such a function be invoked for debugging without running the test case from the start? The answer is from a unit test or endpoint.

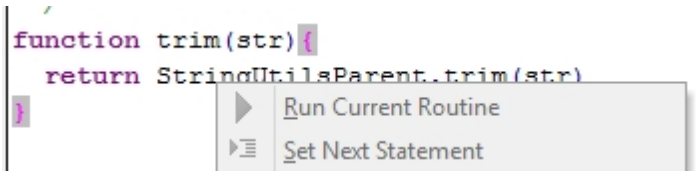
In TestComplete some functions can be run directly from the <Right-Click> menu. These functions must have no parameters. To test a function with parameters then a function of **<Function Name> + UnitTest** or **<Function Name> + EndPoint** with no parameters is created to call the function under test. The purpose of such functions are:

1. To do a limited amount of set up (such as getting today's date or reading a file to get data to pass in to the function or just initialising some hard coded test parameters)
2. To call the target function
3. To validate the results in code or to provide a point at which a breakpoint can be set and the AUT or result of the target function can be validated manually

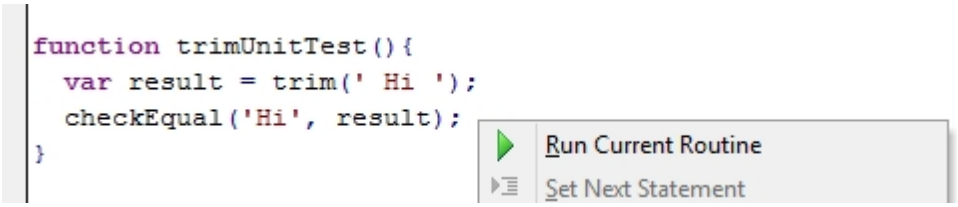
4. To make code self documenting by providing simple usage examples

The examples below illustrate the pattern of a function / UnitTest pair:

UnitTest Example



As the trim function has parameters it cannot be invoked directly from within the TestComplete IDE



The function can be tested by invoking it from the trimUnitTest

EndPoint Example

```
function setFormEndPoint() {
  function doKeys(uiObj, val){
    uiObj.Keys(val);
  }

  /* assumes you are here: http://support.smartbear.com/samples/
testcomplete11/weborders/Process.asp */
  var container = seekByIdStr('ctl00_MainContent_fmwOrder');
  var data =  {
    ctl00_MainContent_fmwOrder_ddlProduct: 'ScreenSaver',
    Quantity: 1,
    'Price per unit:': 30,
    Discount: 15,
    City: withSetter('Melbourne', doKeys),
    State:'VIC',
    Zip: 3126
  }

  /* a parent object can also be used */
  setForm(container, data);
}
```

From the above it can be seen that the usage pattern for <Function Under Test> + UnitTest and <Function Under Test> + EndPoint are exactly the same. Both provide parameterless functions from which to invoke, observe, test and demonstrate the function under test. Note that in the EndPoint above there is no check on the outcome of the function. The user would run this and observe the AUT to see if it worked.

Difference Between UnitTests and EndPoints

UnitTests, like EndPoints are parameterless functions whose purpose is to enable the invocation of other functions for the purpose of testing and debugging. The only difference, other than the naming conventions, is that UnitTests are required to be *self sufficient*. They can read data from a testData file or assign variables or similar actions before calling the target function but the UnitTest is expected to do all the set up by itself. It should also be able to do this on any client machine without requiring any other resources other than the TestComplete project (including browsers, the internet or the AUT) or any configuration. As a consequence of these constraints UnitTests are only usually created for utility functions which do low level data manipulation (e.g. as formatting or validating stings, date calculations). Any function that interacts with the AUT will need to be run from an EndPoint as such a function requires an application to be installed or external service to be running before it can be invoked.

Guidelines for UnitTests and EndPoints

1. **Keep Functions Short and Separate Navigation and Non-Navigational Code**

   To effectively use end points for unit tests, these functions must be readily repeatable. This being the case it is essential that the functions used to compose an automation script are small and that the functions that invoke navigation in the AUT are in separate functions to those that perform other parts of the automation such as data entry and recording the state of the application. If, for example, a function which clicks on a button to open a screen, completes a web form then clicks <Save> this function should be split into three sub-functions as follows:

   ```
   navigateToForm();
   populateForm(params);
   closeForm();
   ```

   By breaking the navigation away from the data entry this enables the data entry to be tested repetitively in isolation from an EndPoint. If these steps were all in one function then the user would have to manually reset the application under test prior to each run of the endpoint

2. **Provide Self Documenting Checks if the Target Function is to be Reused**

   Use of the check functions provided by the framework (see Validation with Check) to validate the result of the function under test. These serve the dual purpose of enabling check to be rerun if the function gets modified at a later date and provide documentation for other users of the target function.

```
function padRightUnitTest() {
  var result;
  result = padRight("loremipsum", "PIE", 15);
  checkEqual('loremipsumPIEPIEPIEPIEPIE', result);

  result = padRight("text", " ", 8);
  checkEqual("text    ", result);

  result = padRight("myawesometext", "!", 20);
  checkEqual("myawesometext!!!!!!!", result);

  result = padRight("aaaaa", "b", 6);
  checkEqual("aaaaab", result);
}
```

3. **Comment EndPoints if Set Up is Not Obvious**

   Because EndPoints required AUT to be in particular state before they are launched, it is often useful to add a short comment at the top of the endpoint describing what state the application should be in prior to launching function.

```
function checkExistsInPageEndPoint(){
  // http://support.smartbear.com/samples/testcomplete10/weborders/
  var result = checkExistsInPage({ObjectType: 'Panel', ObjectIdentifier: 1,
ContentText: '*Clare Jefferson*' });
  check(result);
```

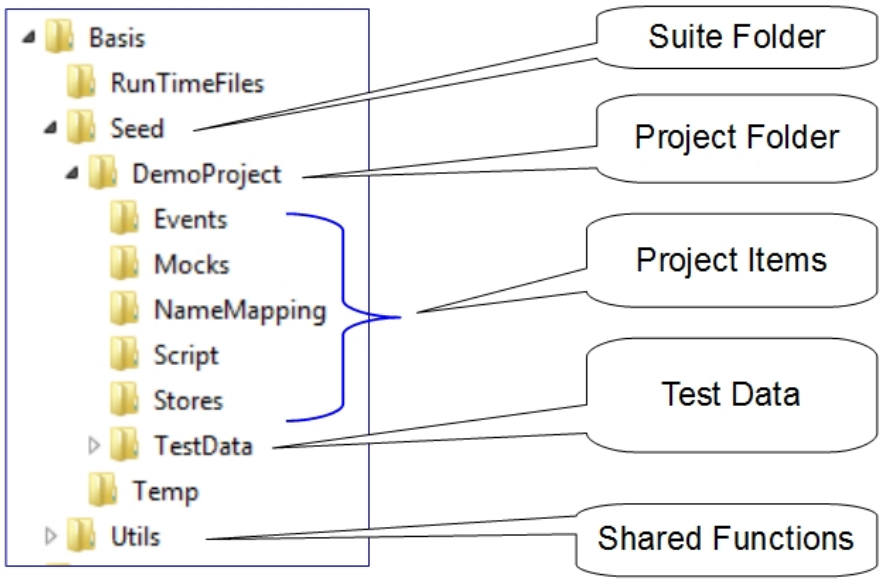
4. **Use Multiple Calls to the Target Function if Required**

   As can be seen from the examples above, it standard practice to add multiple examples and checks to one UnitTest or EndPoint. This tests the target function more thoroughly and provides more comprehensive examples for the benefit of later users. Note that this differs from best practice for the use unit testing frameworks in development environments where one assertion per test is recommended. The reason for this is that the purpose of unit test within unit testing frameworks and EndPoints and UnitTests within the Zenith Framework are different.

5. **Use TestData or TempData**

   Target functions sometimes require complex data such as large JavaScript objects or whole text files as parameters. In such cases it is recommended that utility functions for saving and restoring test data as described in the section below are used.

Using Test and Temp Data



The above illustrates the standard directories of a Zenith project. Of particular importance to UnitTests and EndPoints in the TestData directory and the Temp directory.

The TestData directory is under version control and it is the place where JavaScript objects or text files can be saved checked in and reused from within UnitTest or EndPoint. The Temp directory is not under version control. This directory is used for save large objects or text so that they can be viewed more easily than from within the debugger. The temp directory can also be used to save data for one off debugging purposes but as this will not be checked into version control it will not be able to be reused at a later date. The functions used to save to and restore data from these directories are illustrated below:

```
stringToTestData(lotsOfText, 'myTestFile.txt');
```

Saves a text file to TestData directory => .\TestData\myTestFile.txt is created. If the extension is omitted then it is defaulted to .txt

```
var lotsOfText = testDataString('myTestFile.txt');
```

Reads a text file from the TestData directory => .\TestData\myTestFile.txt is read. If the extension is omitted then it is defaulted to .txt

```
stringToTemp(lotsOfText, 'myTestFile.txt');
```

Saves a text file to Temp directory => ..\Temp\myTestFile.txt is created. If the extension is omitted then it is defaulted to .txt

```
var lotsOfText = tempString('myTestFile.txt');
```

Reads a text file from the Temp directory => ..\Temp\myTestFile.txt is read. If the extension is omitted then it is defaulted to .txt

```
toTestData(myObject, 'myObject.json');
```

Saves a JavaScript object to TestData directory => .\TestData\myObject.json is created. If the extension is omitted then it is defaulted to .json

```
var myObject = fromTestData('myObject.json');
```

Reads a JavaScript object from the Temp directory => .\Temp\myObject.json is read. If the extension is omitted then it is defaulted to .json

```
toTemp(myObject, 'myObject.json');
```

Saves a JavaScript object to Temp directory => ..\Temp\myObject.json is created. If the extension is omitted then it is defaulted to .json

```
var myObject = fromTemp('myObject.json);
```

Reads a JavaScript object from the Temp directory => .\Temp\myObject.json is read. If the extension is omitted then it is defaulted to .json

```
toTempReadable(myObject, 'myObject.txt');
```

Saves a JavaScript object to Temp directory but reformats it to an easily human readable format => .. \Temp\myObject.txt is created. If the extension is omitted then it is defaulted to .txt. Note: there is no function to read this object back. It is intended as an aid for analysis only.

Naming Conventions

The primary naming convention for UnitTests and EndPoints (as described above) is; *targetFunction* => *targetFunctionEndPoint* or *targetFunctionUnitTest*.

Occasionally, for a heavily overloaded function it might be clearer to have more than one EndPoint of UnitTest for each function to separate out tests for the different overloads. In such cases use the naming convention <Target Function> + '_' + <Some Description> + [EndPoint || UnitTest]. A sample of these naming conventions can be seen below:

```
function myCoolFunction(objectOrString){
//...
}

function myCoolFunctionEndPoint(){
//...
}

function myCoolFunction_UsingObjectEndPoint(){
//...
}

function myCoolFunction_UsingStringUnitTest(){
//...
}
```

Unit Placement

Code completion is an important productivity feature of TestComplete. Code completion is, however, less effective when there is a lot of clutter caused by irrelevant functions being displayed and if a script unit is cluttered with such functions then it is also generally harder to navigate. As UnitTests and EndPoints are never called form other functions they can be considered to be a potential source of code clutter. This leads to the general principle than if a script unit will be used by more than one other script unit then UnitTests and EndPoints should be placed in a separate script file. See EndPoints in the Auxiliary Units section for more details.

TestCaseEndPoints - A Special Kind of EndPoint

As described in Test Cases and Test Items a test case comprises of two distinct phases an interactor and a validators phase. There can be many iterations of a single test case based on the number of items returned from the `testItems` function within the same test script. Prior to any test iteration running the environment must be reset and the AUT must be initialised as discussed in Restarts.

The purpose of a `testCaseEndPoint` is to provide a means of easily configuring and invoking a single iteration or all iterations of a test case for development and debugging. When a `testCaseEndPoint` is invoked the framework ensures that all parts of the test: initialising the environment and AUT, invoking the `testItems` function, selecting the target item then running `interactor` and `validators` functions are executed as they would be in a test run.

```
function testCaseEndPoint(){
  var testParams = {
                    testConfig: TEST_CONFIGURATION,
                    itemSelector: null,  //1, lastItemWithValidators, all,
{otherProp: ???}, topIssue
                    country: 'New Zealand',
                    mocking: true
                   };
  runTestCaseEndPoint(testParams);
}
```

Above is an example of a typical `testCaseEndPoint`. As can be seen this is simply a declaration of testParams and an invocation of the framework function `runTestCaseEndPoint`.

Parameters

The `testParams` object includes two properties specific to running the endpoint: testConfig and `itemSelector`.

The `testConfig` always points to the test configuration object within the same script file so it will always be set to: `TEST_CONFIGURATION`. The framework uses this item to identify which test case to run.

The `itemSelector` is used to specify which test item to run. Item selectors are described in detail below.

All other parameters form overrides for runConfig properties that will be used to run the test iteration. In the above example the test case will be run with the default runConfig but the `country` and mocking property will be overridden. Any valid runConfig properties can be overridden in a `testCaseEndPoint` as required for development or debugging. More details about runConfigs can be found in Test Runs.

Item Selectors

The `itemSelector` property determines which one of the test items will be run by the `testCaseEndPoint`. The definition for each of these is described below and refers to the following example:

```
function testItems(runConfig){
  return [
          {
            id: 1,
            toDo: 'add when then',
            message: 'item 0',
            validators: demo_validation
          },
          {
            id: 2,
            when: 'I do this',
            then: 'that happens',
            message: 'item 1',
            validators: demo_validation
          },
          {
            id: 3,
            message: '2 vals',
            validators: [
                          demo_validation,
                          another_validation
                        ]
          },
          {
            id: 4,
            when: '3',
            then: '33',
            message: 'item 2'
          }
        ];
}
```

`itemSelector: null`
: the last item returned from the testItems function will be run. This being the case the developer will always be working on the last iteration appended to the testItems array - *in the above example item id 4 would be run*

`itemSelector:` 3
: if the itemSelector is set to a single value then the test item with the matching id will be run - *in the above example item id 3 would be run*

`itemSelector:` lastItemWithValidators
: lastItemWithValidators is actually a function. This value will result in the selection of the item before the first test item with a null or undefined validators property. This is useful when working with mock data and developing validators for each item from the first to the last item. See Test Development Workflow for more details on using mock data in developing tests - *in the above example item id 3 would be run as item id 4 is the first test item with no validators*

`itemSelector: all`
: all will result in every item returned by the `testItems` function being run - *in the above example all of items 1, 2,3 and 4 would be run*

`itemSelector: {then: 'that happens'}`
when a JavaScript object is used as a selector then the first item matching all the properties of the selector is run *- in the above example item id 2 would be run*

`itemSelector: topIssue`
after a test case is run as part of a test run, or after a `testCaseEndPoint` with `itemSelector: all` is run, a summary of any test items with errors or a toDo property will be written to a text file <Test Case Name> + _ Issues.txt in the same directory as the TestComplete log (e.g. `C:\TestCompleteLogs\10_07_2015_10_42 PM_50_801\`). So after running Demo_Test with errors or toDo items then a Demo_Test_Issues.txt file will be created. An example of the contents of such a file is included below:

```
[
    includeReason: TO DO
    test item:
            id: 1
            toDo: add when then
            message: item 0
    application state:
            message: stuff
    validation [item id: 1]:
            demo validation: passed
    ----------------------------------------
    includeReason: ERRORS / WARNINGS AND TO DO
    test item:
            id: 4
            when: 3
            then: 33
            message: item 2
    application state:
            message: more stuff
            interactor errors: dummy interactor error
    validation [item id: 4]: Item Has No Validations
]
```

The `topIssue` selector will simply run the id of the top item in this summary. This enables the user to open this text file and use the workflow:

1. resolve the error or complete the toDo item
2. delete the item from the top of the text file save
3. rerun (the item associated with the new top issue would then be picked up)

In the above example item id 1 would be run then when that item is deleted from Demo_Test_Issues.txt item id: 4 would be run

Invocation

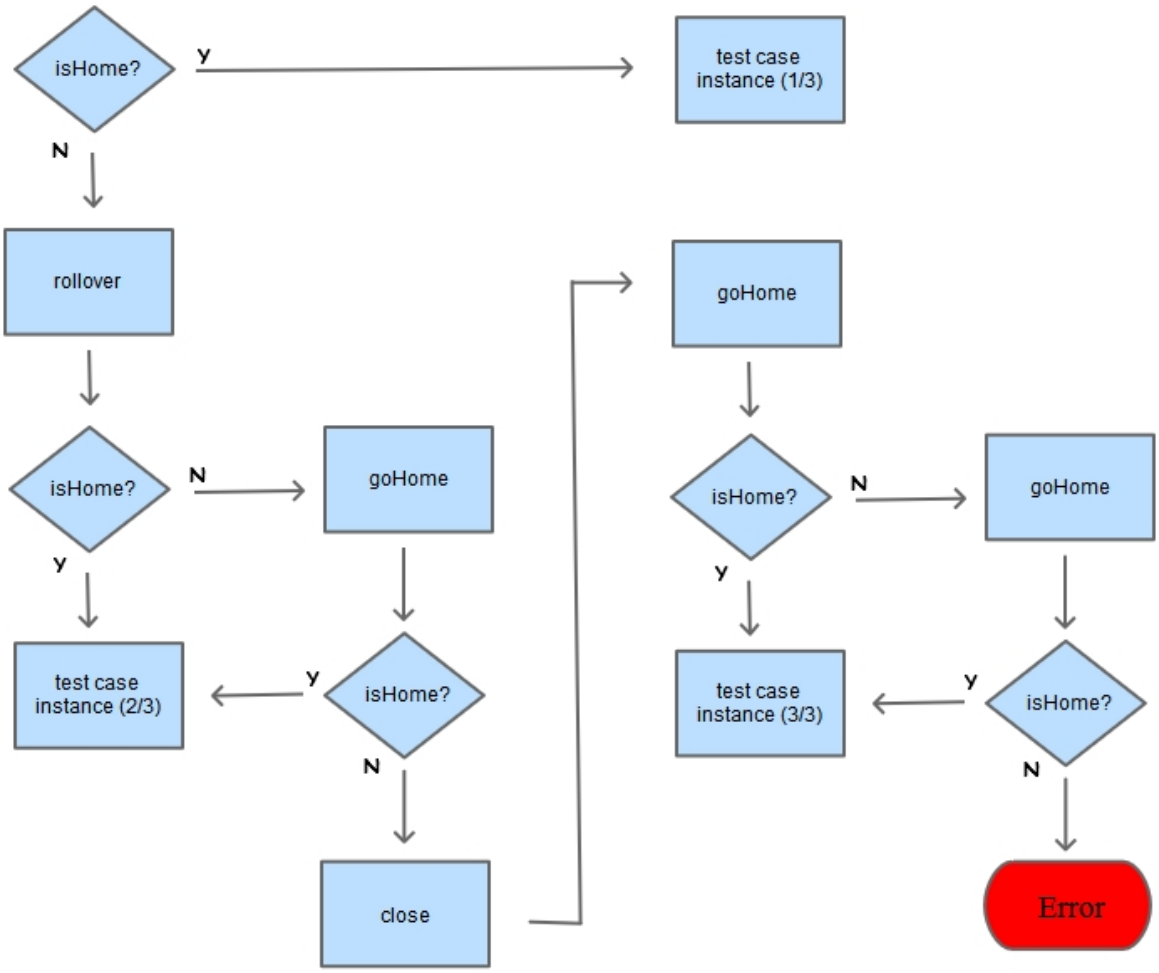
As with any UnitTest or EndPoint invoking a testcaseEndPoint is simply a matter of <right click><Run> but in addition to this method of invocation the framework has been set up to run the last testCaseEndPoint as the default project item. This means that from anywhere within the project you can rerun the last testCaseEndPoint by using the standard <Run Project> TestComplete shortcut: <Ctrl+F5>.

Restarts

Normally prior to running a `testCaseEndPoint` the associated restart functions are run in the similar manner as when a batch of tests is run as part of a test run, i.e. the test environment, such as databases, will be reset (a.k.a. rollover) and AUT will be taken to its *"Home State"*. There is, however, one important difference. If the AUT already in its *"Home State"* (as defined by the isHome function in the associated restart script) then no rollover will occur.

For example if the home page after logging in is deemed to be the *"Home State"* for an online shopping

application then the developer could navigate to this page prior to running the `testCaseEndPoint` and no rollover would occur. If a rollover is required (e.g. due to a need to refresh the database) then simply closing the browser, hence taking the AUT out of its *"Home State"* will cause the rollover to be run next time the `testCaseEndPoint` is invoked.



The restart workflow prior to executing the test phases of a testCaseEndPoint is almost identical to that prior to running a batch of tests in a test run. Details of restarts and this workflow can be found in Restarts.

Mocking

As described in Test Development Workflow the use of mock files enables the development of business rule validation independently of any interaction of the AUT. During a test run a mock file is saved to the Mocks directory for every iteration of the test. These files contain the apState object which summarises the state of the AUT and it is this object that is validated by the functions assigned to the validators property of each test item.

If the testParams object in the `testCaseEndPoint` has mocking enabled: `mocking: true` and the mock file exists for the target item id, then when the `testCaseEndPoint` is run the restart functions (as described in the previous section) will be skipped as will the interactor. Only the validators will be run against the previously saved mock data.

This means that `testCaseEndPoint` can be used to develop business rule validation separately from or even before the AUT has been developed. More details on these alternate workflows can be found in: Test Development Workflow.

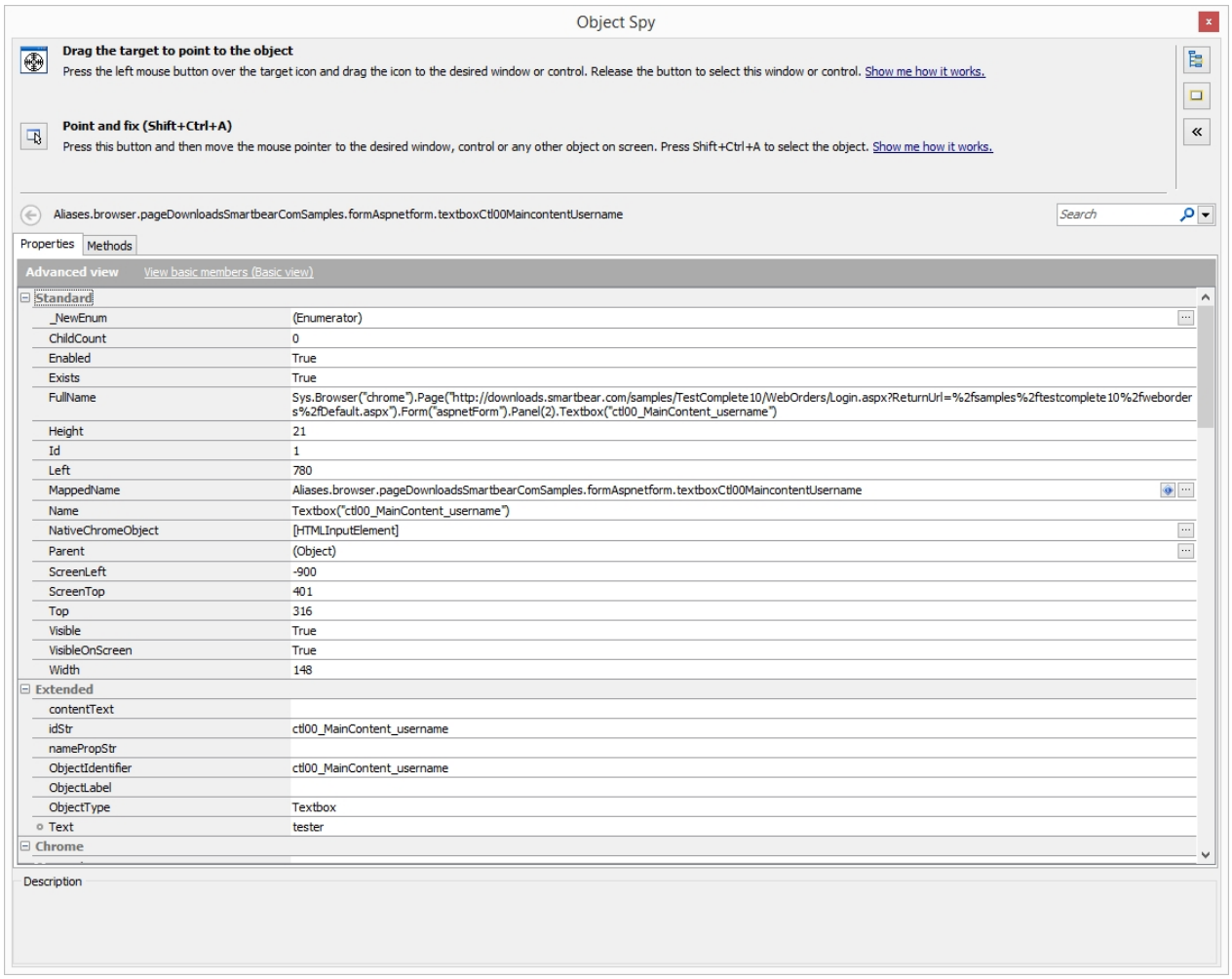
Interacting with the UI

The UI interaction part of an automated UI test requires the testing software to locate an object in UI (such as a button) and then interact with that object e.g. reading the value of its `<enabled>` state for later validation or clicking on it. Test Complete's Name Mapping provides facilities for finding objects but UI interaction in Zenith uses a more flexible and robust mechanism based on functions provided by TestComplete `FindChild` and `FindAllChildren`. When initially exploring an application it is easy to record a script which will use NameMapping but once the exploratory phase of your test development is over you should rely on the functions provided by the Zenith Framework.

Properties

Finding a UI object, whether using built in TestComplete functions or Zenith Framework functions is always a matter of comparing search criteria properties (such as an id string) with the properties of a particular UI object. UI object properties can be inspected from within the TestComplete IDE by invoking the object spy:  . More information about using the object spy can be found in TestComplete Help. An example of the object spy in use is shown below.

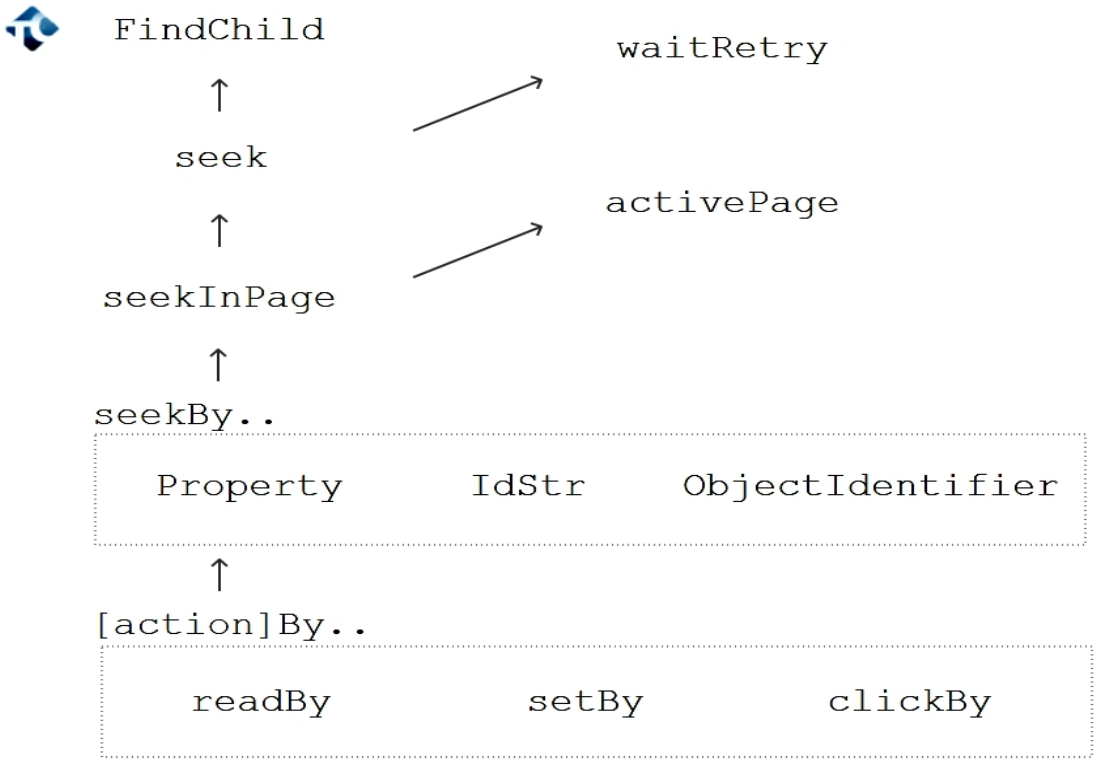
The sample SmartBear login page as it appears in the browser



The user name text box as it appears in the object spy. This will be used as a reference object in the following examples

Basic UI Interaction Function Hierarchy

As mentioned above, Zenith UI interaction functions are built on `FindChild` and `FindAllChildren`, two functions provided by TestComplete. There are low level functions to find a UI object or objects based on given criteria and built on these are convenience find functions that find objects base on single or common properties. Built on these convenience functions are functions that perform common actions such as reading or setting the value of the object. Beyond this there are power functions for setting forms with minimal code or interacting with HTML tables. Note that at the time of writing all but the `seek` function deal with browser based applications only. Similar function hierarchies, however, could be developed for other applications such as mobile or fat client applications as the need arises. The framework functions that act on singular objects are discussed below (collection functions are discussed in later sections):



The function hierarchy for singular UI functions. Arrows represent a "uses" relationship

Finder Functions

findChild

`findChild` is the TestComplete function at the root of the hierarchy. Given a UI object, such as a browser, a call to `findChild` will return the first object within its child objects that matches the provided criteria. The search for this object can be fully recursive or the depth of the search can be limited. If a matching object is found then it will be returned, if no matching object is found then a stub object will be returned with a single `Exists` property that will be `false`.

The following example is a slightly modified version of the example shown in TestComplete Help

```
function Test()
{
  var PropArray, ValuesArray, p, w;

  // Creates arrays of property names and values
  PropArray = new Array("WndCaption", "Visible");
  ValuesArray = new Array("Font style:", true);

  // Searches for the window
  p = Sys.Process("Notepad");
  w = p.FindChild(PropArray, ValuesArray);

  // Processes the search results
  if (w.Exists)
    Log.Message(w.FullName);
  else
    Log.Error("The object was not found.");
}
```

The above code is searching an open Notepad application for an object that has a windows caption of "Font style" and is currently visible.

Although the above code is functional it is not very user friendly or readable. The main problem stems from the need to create a new VB Script style of array and then match up the positions of properties and values in these arrays to understand what properties are being matched to what values. When using the Zenith Framework you never need to use this function directly. Instead a much more JavaScript and user friendly wrapper around FindChild is provided: `seek`.

seek

`seek` is at the root UI interaction function of the Zenith Framework. It serves a similar role to `findChild` but as can be seen by the example below is a lot easier to use and read.

```
function test(){
  // Searches for the window
  var p = Sys.Process("Notepad");
  var w = seek(p, {
                    WndCaption: "Font style:",
                    Visible: true
                  });

  // Processes the search results
  var message = hasValue(w) ? w.FullName : "The object was not found."
  log(message);
}
```

The most noticeable difference when comparing the examples above is that the criteria in the second example are passed in to the function as a JavaScript object, rather than a pair of arrays.

There are also powerful features available with `seek` that are not implemented in `findChild`. These features are explained below:

The following is a sample of valid calls to seek:

```
seek(parentObj, criteria)
seek(parentObj, criteria, timoutMs)
seek(parentObj, criteria, timoutMs, depth)
seek(parentObj, criteria, criteria, criteria, timoutMs)
seek(parentObj, criteria, criteria, criteria, timoutMs, depth)
```

The parameters are as follows:

`parentObj - required:`
: An automation object obtained directly by name e.g. `Sys.Browser("chrome")` or as the result of some other seek operation. This is the parent under which the search will take place (i.e the search will return a matching child under this parent if it exists)

`criteria* - required:`
: JavaScript objects or functions which need to be matched. For JavaScript object criteria the first UI object that matches all the properties listed in the JavaScript object is returned. Wild cards can be used for string properties of JavaScript objects (e.g. `contentText: "Log*out"`). As well as JavaScript objects, functions that take a UI object as a parameter and return a Boolean can also be used as criteria but this feature is really only needed for edge cases where the criteria cannot be fully described in an object

  Multiple (any number of) criteria can be used. When multiple criteria are used `seek` will first find an object matching the first criteria then the children of this UI object will be searched for an object matching the second criteria and so on. The reason for using multiple criteria is to speed up execution time in very large UIs. For example, on a huge massively nested web page seeking a table cell deep in the DOM might be quite slow. If, however, the table is known to be in the bottom panel then specifying criteria to find bottom panel first will mean that page elements in the top and middle panels wont be searched and the target object will be found more quickly.

`timoutMs - optional [default: 10000]`
: When a integer is in the arguments list the first number will be used as a time out (in ms). This means that if the object is not found `seek` will keep re-executing the search for approximately the set time before returning the stub object with `{Exists: false}`. Seek will always search at least once, i.e. there is no time out check until after the first search attempt is complete. The implementation of the time out uses the framework function `waitRetry` which is discussed in Waiting on the AUT.

`depth - optional [default: 10000]`
: When two integers are provided in the arguments list the second refers to how deep within the parent object's children the `seek` function will search. So if there is a button within a form, inside an inner panel, in an outer panel in a web page (4 levels in from the page) then seek will only find this button if the `depth` is 4 or more. When specifying `depth` then `timoutMs` must also be specified because depth is identified as the second number.

  Using `depth` and `timoutMs` is particularly useful when there is need to detect one or more states in a UI object without stalling the automation. For example a shared function that clicks a login image button on one type of web page and a login link on another might need to seek a link first then if the link is not found search for the button. If the default values were used then the automation would retry for ten seconds searching for a link before searching for a button. Limiting the depth and the time out solves this problem.

seek Variations - Full Example

```
function testSeek(){
  var browser = Sys.Browser("chrome");

  //simple
  var userNameTxt = seek(browser, {idStr: 'ctl00_MainContent_username'});
  highlight(userNameTxt);

  // find the text box within its parent panel - this would not
  // be useful on such a small web page as simple search is very fast
  userNameTxt = seek(browser, {
                                    ObjectType: 'Panel',
                                    className: 'log*n',
                                    Visible: 'True'
                                  },
                                  {idStr: 'ctl00_MainContent_username'});
  highlight(userNameTxt);

  // setting the time out to 0 means the search will only run once if the
object is not found
  // before searching a second time the time out will be exceeded and the
search aborted
  userNameTxt = seek(browser, {idStr: 'ctl00_MainContent_username'}, 0);
  highlight(userNameTxt);

  // limiting the search depth means that this search will fail
  // and return the stub object (the user name text box is nested more than
two levels from
  // under the browser)
  userNameTxt = seek(browser, {idStr: 'ctl00_MainContent_username'}, 0, 2);
  checkFalse(userNameTxt.Exists);

  //using a function instead of an object - rarely needed in practise
  function isUserTxtBox(uiObj){
    return uiObj.idStr === 'ctl00_MainContent_username';
  }
  userNameTxt = seek(browser, isUserTxtBox);
  highlight(userNameTxt);
}
```

Once you understand how `seek` works then understanding the web convenience functions that are based on seek is quite straightforward. They are simply shortcut variants of `seek` that provide default parameters or execute default behaviour after the UI object is returned.

seekInPage

`seekInPage` is a convenience function for web page automation. It behaves exactly the same as seek except the active web page (i.e. active tab) for the target browser is substituted for the parent object. This is implemented as a call to `activePage()`, to obtain the active web page, followed by a call to seek. The parameters are the same as seek except there is no `parentObj`. The following are all valid calls to `seekInPage`:

```
  seekInPage(criteria)
  seekInPage(criteria, timoutMs)
  seekInPage(criteria, timoutMs, depth)
  seekInPage(criteria, criteria, criteria, timoutMs)
  seekInPage(criteria, criteria, criteria, timoutMs, depth)
```

seekInPage - Full Example

```
function testSeekInPage(){
  //simple
  var userNameTxt = seekInPage({idStr: 'ctl00_MainContent_username'});
  highlight(userNameTxt);

  // specifying parent
  userNameTxt = seekInPage({
                            ObjectType: 'Panel',
                            className: 'log*n',
                            Visible: 'True'
                           },
                           {idStr: 'ctl00_MainContent_username'});
  highlight(userNameTxt);

  // setting the time out to 0 means the search will only run once
  userNameTxt = seekInPage({idStr: 'ctl00_MainContent_username'}, 0);
  highlight(userNameTxt);

  // limiting the search depth means that this search will fail as text box
more than 2 deep
  userNameTxt = seekInPage({idStr: 'ctl00_MainContent_username'}, 0, 2);
  checkFalse(userNameTxt.Exists);

  //using a function instead of an object
  function isUserTxtBox(uiObj){
    return uiObj.idStr === 'ctl00_MainContent_username';
  }
  userNameTxt = seekInPage(isUserTxtBox);
  highlight(userNameTxt);
}
```

Other than the comments being abbreviated, this example is the same as for seek but it is simpler as there is no need to pass a browser or page object into the function.

seekByProperty

A frequent use case is to seek an object in a web page by a single property only. In such cases the need to define a single property criteria object can be avoided by using `seekByProperty`. The parameters are the same as for seek in page but the last criteria object is replaced by two parameters: `propertyName` and `propertyValue` which are added to the end of the parameter list. `seekByProperty` is implemented by simply by slicing off the last two parameters and creating a criteria object from them then calling `seekInPage`.

The following are all valid calls to `seekByProperty`:

```
  seekByProperty(propertyName, propertyValue)
  seekByProperty(timoutMs, propertyName, propertyValue)
  seekByProperty(criteria, timoutMs, depth, propertyName, propertyValue)
  seekByProperty(criteria, criteria, criteria, timoutMs, propertyName,
propertyValue)
  seekByProperty(criteria, criteria, criteria, timoutMs, depth, propertyName,
propertyValue)
```

seekByProperty - Full Example

```
function testSeekByProperty(){
  //simple
  var userNameTxt = seekByProperty('idStr', 'ctl00_MainContent_username');
  highlight(userNameTxt);

  // specifying parent
  userNameTxt = seekByProperty({
                              ObjectType: 'Panel',
                              className: 'log*n',
                              Visible: 'True'
                            },
                            'idStr', 'ctl00_MainContent_username');
  highlight(userNameTxt);

  // setting the time out to 0 means the search will only run once
  userNameTxt = seekByProperty(0, 'idStr', 'ctl00_MainContent_username');
  highlight(userNameTxt);

  // limiting the search depth means that this search will fail as text box
more than 2 deep
  userNameTxt = seekByProperty(0, 2, 'idStr', 'ctl00_MainContent_username');
  checkFalse(userNameTxt.Exists);
}
```

Compared to seekInPage the last criteria object has been replaced by the property name / value (idStr and ctl00_MainContent_username)

seekByIdStr

If a developer has assigned a stable id via the id css attribute on the web page element then this is the most convenient robust property to use to find that element. These ids are also self documenting when sensible naming conventions have been adopted. TestComplete provides the `idStr` property to access this id. `seekByIdStr` is the same as `seekByProperty` in that it starts searching from the active web page but instead of `propertyName` and `propertyValue` it just requires the `idStr` value as the last parameter. `seekByIdStr` is implemented by calling `seekByProperty` with the `propertyName` of `'idStr'` and propertyValue of the `idStr` passed in.

The following are all valid calls to `seekByIdStr`:

```
  seekByIdStr(idStr)
  seekByIdStr(criteria, idStr)
  seekByIdStr(timoutMs, idStr)
  seekByIdStr(criteria, timoutMs, depth, idStr)
  seekByIdStr(criteria, criteria, criteria, timoutMs, idStr)
  seekByIdStr(criteria, criteria, criteria, timoutMs, depth, idStr)
```

seekByIdStr - Full Example

```
function testSeekByIdStr(){
  //simple
  var userNameTxt = seekByIdStr('ctl00_MainContent_username');
  highlight(userNameTxt);

  // specifying parent
  userNameTxt = seekByIdStr({
                             ObjectType: 'Panel',
                             className: 'log*n',
                             Visible: 'True'
                            },
                            'ctl00_MainContent_username');
  highlight(userNameTxt);

  // setting the time out to 0 means the search will only run once
  userNameTxt = seekByIdStr(0, 'ctl00_MainContent_username');
  highlight(userNameTxt);

  // limiting the search depth means that this search will fail as text box
more than 2 deep
  userNameTxt = seekByIdStr(0, 2, 'ctl00_MainContent_username');
  checkFalse(userNameTxt.Exists);
}
```

seekByObjectIdentifier

If no `idStr` has been assigned to a web element then TestComplete generates a random idStr (see TestComplete Help for more details). In such cases an alternative property, the `ObjectIdentifier` may be more stable and descriptive and as such may be a better property with which to find a UI object. `seekByObjectIdentifier` is implemented and used in almost exactly the same way as `seekByIdStr`. The only difference is the target UI element property is different `(ObjectIdentifier` as opposed to `idStr)`.

The following are all valid calls to `seekByObjectIdentifier`:

```
  seekByObjectIdentifier(objectIdentifier)
  seekByObjectIdentifier(criteria, objectIdentifier)
  seekByObjectIdentifier(timoutMs, objectIdentifier)
  seekByObjectIdentifier(criteria, timoutMs, depth, objectIdentifier)
  seekByObjectIdentifier(criteria, criteria, criteria, timoutMs,
objectIdentifier)
  seekByObjectIdentifier(criteria, criteria, criteria, timoutMs, depth,
objectIdentifier)
```

seekByObjectIdentifier - Full Example

Note: In the example `idStr` and `ObjectIdentifier` are the same so the only difference between this example and that for `seekByIdStr` is the function name.

```
function testSeekByObjectIdentifier(){
  //simple
  var userNameTxt = seekByObjectIdentifier('ctl00_MainContent_username');
  highlight(userNameTxt);

  // specifying parent
  userNameTxt = seekByObjectIdentifier({
                                          ObjectType: 'Panel',
                                          className: 'log*n',
                                          Visible: 'True'
                                       },
                                       'ctl00_MainContent_username');
  highlight(userNameTxt);

  // setting the time out to 0 means the search will only run once
  userNameTxt = seekByObjectIdentifier(0, 'ctl00_MainContent_username');
  highlight(userNameTxt);

  // limiting the search depth means that this search will fail as text box
more than 2 deep
  userNameTxt = seekByObjectIdentifier(0, 2, 'ctl00_MainContent_username');
  checkFalse(userNameTxt.Exists);
}
```

Reader Functions

There is not much point for a script to find a UI element if it doesn't do anything with it. One of the most common interactions between a script and UI element is to read its default value property such as the text of a label or the checked state of a checkBox. The Zenith Framework provides a function to read the default property of a given UI object called `read`. There are analogous functions to the above `seek*` functions that take exactly the same parameters and find the object in exactly the same way except that instead of returning the object it returns the default value of the object. If the object is not found then an "`Object does not exist`" exception is thrown. Reader functions are implemented by calling the analogous `seek` function then calling `read` on the UI object returned.

All the examples above for `seek*` would work equally well for read*. This being the case only limited simple reader* examples are provided below. All these examples would read the value from the user name text field.

read

```
var browser = Sys.Browser("chrome");
var textInField = read(browser, {idStr: 'ctl00_MainContent_username'});
```

Reads the value of the matching object given the parent

readInPage

```
var textInField = readInPage({idStr: 'ctl00_MainContent_username'});
```

Reads the value of the matching object in the active page

readByProperty

```
var textInField = readByProperty('idStr', 'ctl00_MainContent_username');
```

Reads the value of an object within the active page matching a single property

readByIdStr

```
var textInField = readByIdStr('ctl00_MainContent_username');
```

Reads the value of an object within the active page of the given `idStr`

readByObjectIdentifier

```
var textInField = readByObjectIdentifier('ctl00_MainContent_username');
```

Reads the value of an object within the active page of the given `ObjectIdentifier`

Setter Functions

For setting a small number of objects setter functions can be used. If a large number of objects are being populated on a web page then `setForm`, as described below, is a normally better option.

Setter functions work in a similar way to reader functions but instead of reading the default value they use the framework's base `set` function to set the value of the UI object. They have exactly the same params as the base `seek` function but for a single trailing parameter which is the value the object is to be set to (e.g. `true` or `false` for a checkbox or text for a textbox).

For example the following is a valid call to `setInPage`:

```
setInPage({
            ObjectType: 'Panel',
            className: 'log*n',
            Visible: 'True'
           },
           {idStr: 'ctl00_MainContent_username'},
           5000,
           4,
           'Tester');
```

The above will find an object with the `idStr`: `ctl00_MainContent_username` *within the active page and inside a panel with a* `className` *of* `'log[any chars]n'`. *It will keep retrying for 5 seconds if the object is not found straight away and will only search to a depth of 4 under the active web page. When the object is found the text of the text box will be set to* `'Tester'`. *If we wanted to read the value of this text box the same example would work with* `readInPage` *excluding the last parameter. Note that the complexity of the above example is caused by the optimisations (parent object specification, non default time out and depth limitation). The need for these optimisations are edge cases and setting objects can normally be expressed far more simply as can be seen in the below examples.*

set

```
var browser = Sys.Browser("chrome");
set(browser, {idStr: 'ctl00_MainContent_username'}, 'Tester');
```

Sets the value of the matching object given the parent

setInPage

```
setInPage({idStr: 'ctl00_MainContent_username'}, 'Tester');
```

Sets the value of the matching object in the active page

setByProperty

```
setByProperty('idStr', 'ctl00_MainContent_username', 'Tester');
```

Sets the value of an object within the active page matching a single property

setByIdStr

```
setByIdStr('ctl00_MainContent_username', 'Tester');
```

Sets the value of an object within the active page of the given `idStr`

setByObjectIdentifier

```
setByObjectIdentifier('ctl00_MainContent_username', 'Tester');
```

Sets the value of an object within the active page of the given `ObjectIdentifier`

Click Functions

In addition to reading and setting, it is common requirement to click a UI object. `click*` functions follow the exact same pattern as the above reader and setter functions with addition one special function (`clickLink`).
Examples below:

click

```
var browser = Sys.Browser("chrome");
click(browser, {idStr: 'home_link'});
```

Clicks the matching object given the parent

clickInPage

```
clickInPage({idStr: 'home_link'});
```

Clicks the matching object in the active page

clickByProperty

```
clickByProperty('idStr', 'home_link');
```

Clicks an object within the active page matching a single property

clickByIdStr

```
clickByIdStr('home_link');
```

Clicks an object within the active page of the given `idStr`

clickByObjectIdentifier

```
clickByObjectIdentifier('home_link');
```

Clicks an object within the active page of the given `ObjectIdentifier`

clickLink

```
clickLink('Home');
```

Clicks the first link found in the active page with the `ContentText` *property of* `'Home'`

The "h" Suffix

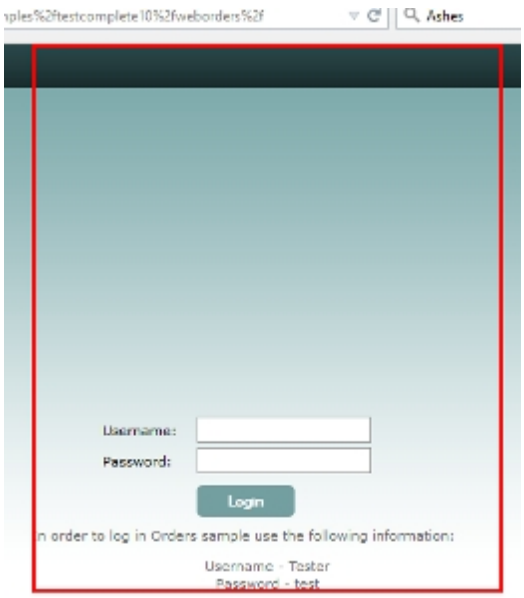
For **all** the functions listed above there is a matching function with the same name but for an "h" suffix e.g:

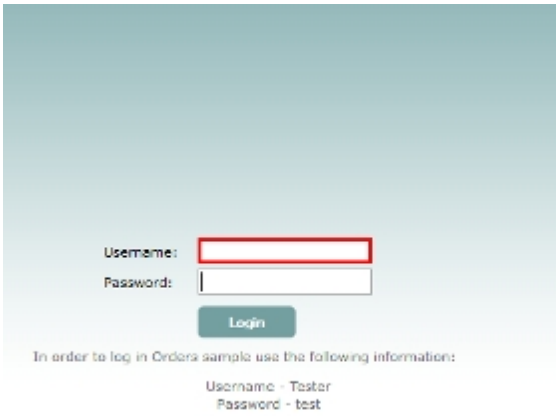
```
seekInPageh
readByPropertyh
setByIdStrh
clickByObjectIdentifierh
clickLinkh
```

These are purely development functions. They work exactly as the above functions but highlight each object on screen as it is found (so if there are 2 parent objects specified in the function then this function will highlight both parent objects and the target object as they are found). These can be useful from time to time to give the developer a quick hint when an object is not found by one of the above functions as expected. Simply append an h to the failing function and rerun it to get an indication of what level the failure occurred. When the issue is resolved then the 'h" can be removed. Example below:

```
seekInPageh({
             ObjectType: 'Panel',
             className: 'log*n',
             Visible: 'True'
            },
            {idStr: 'ctl00_MainContent_username'});
```

The above call will cause the parent panel and then the target text box to be highlighted on screen as they are found





Sometimes Performance is More Important than Robustness

Normally the simplest forms of the above functions will be more than performant enough, but in some edge cases (such as massive web pages) they will not, and there will be a need to specify parent containers and/or limit search depth to reduce the search scope of the function and hence increase its performance. This has been demonstrated in the section above describing `seek`. In rare cases even this may not be fast enough. For example, if there is a text box parented by a pop up panel, that also contains selection options that include every suburb in a country and that text box is named such that it is found last by the seek function within that parent. In this scenario the seek function will need to inspect all the suburb panels before it finds the target panel with the text box. Although using seek functions is robust, in this case the use of such functions could be prohibitively slow. The best option, if the full name of text box is stable, may be to isolate populating this text box in one function and to use the full name to instantly find the object. If developer was to change the object (DOM) hierarchy of the text box then the automation would break but it would be easy to fix by changing the isolated function.

Below is an example of directly addressing an object by full name:

```
var userTextBox = Sys.Browser("chrome").Page("http://downloads.smartbear.com/
samples/TestComplete10/WebOrders/Login.aspx?ReturnUrl=%2fsamples%
2ftestcomplete10%2fweborders%
2f").Form("aspnetForm").Panel(2).Textbox("ctl00_MainContent_username");

//For X Browser testing or if the url of the page was dynamic and for to make
reading easier the following would also work
userTextBox =
activePage().Form
("aspnetForm").Panel(2).Textbox("ctl00_MainContent_username");
```

Collection Functions

For cases where more than one object is being searched for, such as getting a list of all text boxes, then `seekAll` or `seekAllInPage` can be used.

seekAll

Finds all the objects that match a criteria that are children of a given object.

The following is a sample of valid calls to `seekAll`:

```
seekAll(parentObj, criteria)
seekAll(parentObj, criteria, depth)
seekAll(parentObj, criteria, depth, false)
```

Note: Unlike seek, only one criteria can be specified and there is no timeout parameter. `seekAll` will simply search the whole object tree (optionally to a limited depth) and return an array of all the objects that match the given criteria.

The parameters are as follows:

`parentObj - required:`
An automation object obtained directly by name e.g. `Sys.Browser("chrome")` or as the result of some other seek operation. This is the parent under which the search will take place (i.e the search will return all matching children under this parent)

`criteria - required:`
A singular JavaScript object which forms the filter criteria. Wild cards can be used for string properties (e.g. `contentText: "Log*out"`)

`depth - optional [default: 10000]`
The depth to which the object tree is to be searched

`refresh - optional [default: true]`
: Whether to refresh the UI object cache before searching. Rarely used - see TestComplete help for details

The following returns all the TextNodes (a label type of object) in a web form:

```
var form = seekInPage({ObjectType: 'Form'});
var textNodes = seekAll(form, {ObjectType: 'TextNode'});
```

seekAllInPage

Like a cross between `seekInPage` and `seekAll` returns all the objects matching the criteria in the active web page:

The following returns all the TextNodes (a label type of object) in a web page:

```
var textNodes = seekAllInPage({ObjectType: 'TextNode'});
```

The above is similar to the `seekAll` example but there is no need to specify a parent object as the active page is used

seekParent

`seekParent` is an edge case function that does not fit in to any of the categories above. Rather than seeking an object from the parent down through the children, `seekParent` takes a child object as a parameter and will seek up through the parent links until it finds an object that matches a given criteria. This function can be used for optimisation of scenarios such as:

A huge dynamically generated web page has a panel deep in the DOM with a lot of fields that need to be filled out. Also there are no consistent ids on the panels. In such a case it may be a little slow to use the `set*` functions described above due to the need to traverse the DOM for each field. `setForm` described below would be faster but the initialisation stage of executing `setForm` would still be slow. What quicker strategy would be be:

1. Use `seekInPage` to find the first edit UI Object, such as a `textBox`
2. Use `seekParent` to `find` the nearest parent panel
3. Use this parent panel as a base object for the above `set*` functions or `setForm`

The reason the above might be the best solution is seek only needs to traverse from the top to the level of the edit buttons once. As the panel that immediately parents the edit elements would be used as a search base for the other elements these would be located extremely quickly.

The parameters for `seekParent` are as follows:

`childObject - required:`
An automation object obtained as the result of some other seek operation. This is the child to which the target object is a parent.

`criteria - required:`
A singular JavaScript object which forms the filter criteria. Wild cards can be used for string properties (e.g. `contentText: "Log*out"`)

`refresh - optional [default: true]`
Whether to refresh the UI object cache before searching. Rarely used - see TestComplete help for details

Example

```
function seekParentEndPoint(){
  var userNameTxt = seekInPage({idStr: 'ctl00_MainContent_username'}, 0);
  set(userNameTxt, 'Tester');
  var parentPanel = seekParent(userNameTxt, {
                                              ObjectType: 'Panel',
                                              Visible: 'True'
                                            })

  setByIdStr(parentPanel, 'ctl00_MainContent_password', 'test');
}
```

The above demonstrates how `seekParent` could be used to speed up filling in the login page. First the user name field is found then its parent panel is used as a base for setting the password field. Note that this demo site login form is a very small web page so this strategy would not be noticeably quicker than just using plain set functions. The example is for "demonstration only".

Power Functions

`setForm` is the Zenith Framework's power function for populating web pages with minimal code and maximum code readability and run time efficiency. When populating a web page with more than one field `setForm` should be the default strategy. Related to `setForm` is a helper function `withSetter` which enables the default setter for an object to be overridden and a design time convenience function `setFormR` which allows the developer to automatically generate calls to setForm and avoid wasting time with the object spy or a text editor in configuring input parameters.

setForm

`setForm` takes a JavaScript object and sets the data elements in a web page or some other web container object (such as a panel) based on the ids of the editable objects themselves or the text of the nearest "label like" object.

The parameters are as follows:

`container - optional [default: active page]:`
An automation object obtained directly by name e.g. `Sys.Browser("chrome")` or as the result of some other seek operation. This is the parent within which `setForm` will search for editable objects to populate

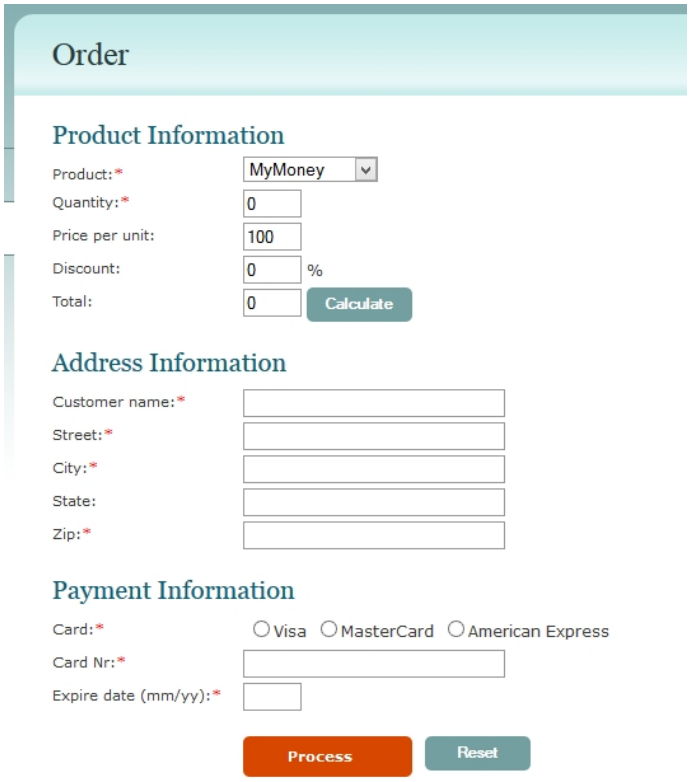
`data - required:`
A JavaScript object which maps:
key: one of an idStr or an ObjectIdentifier of the target editable object or a uniquely identifiable text fragment from a nearby label (as the key) to
value: the value the object is to be set to or a call to withSetter which itself takes a value and a setter function (discussed in more detail below)

`setFunctionOverride - optional [default: null]`
By default `setForm` uses the framework's `set` function to populate the editable objects found. This function can be overridden by passing a function to this parameter.

The following are all valid calls to `setForm`:

```
setForm(data)
setForm(parentObject, data)
setForm(parentObject, data, alternateSetter)
setForm(data, alternateSetter)
```

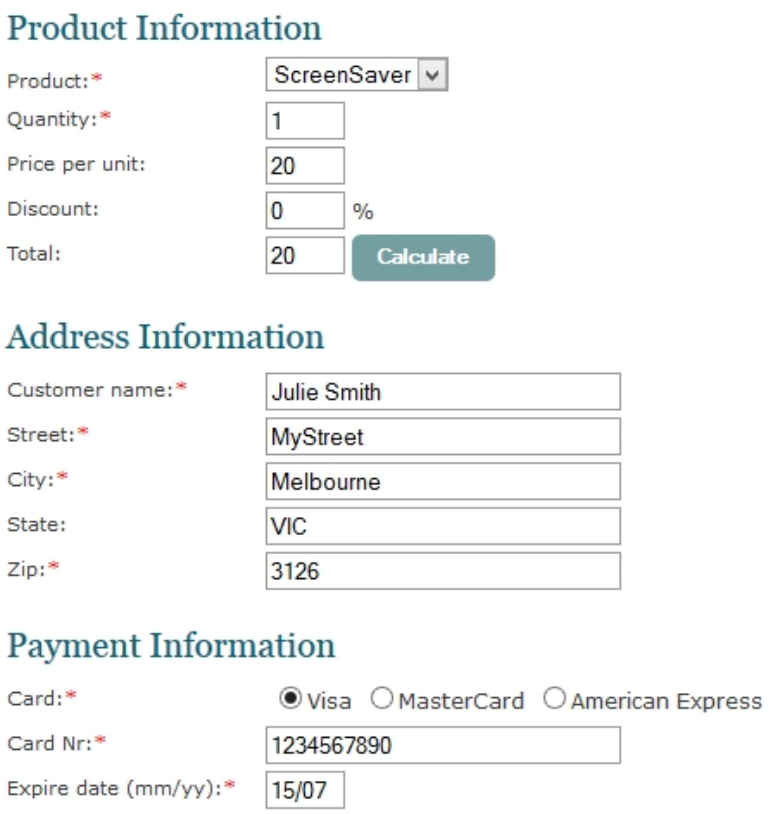
The examples in the following discussion will be based on the Order Page of the SmartBear demo web site:



Basic Example

```
function doKeys(obj, val){
  obj.Keys(val);
}

function setFormEndPoint() {
  //assumes you are here: http://support.smartbear.com/samples/testcomplete11/weborders/Process.asp
  var data =  {
    ctl00_MainContent_fmwOrder_ddlProduct: 'ScreenSaver',
    Quantity: 1,
    'Price per unit:': 30,
    Discount: 15,
    name: 'Julie Smith',
    Street: 'MyStreet',
    City: withSetter('Melbourne', doKeys),
    State:'Vic',
    Zip: 3126,
    Visa: true,
    'Card Nr:': '1234567890',
    'Expire date': '15/07'
  }

  setForm(data);

  //a parent object can also be used (a repeat filling of the form)
  var container = seekByIdStr('ctl00_MainContent_fmwOrder');
  setForm(container, data);
}
```

After executing the above the form would be populated as follows:



Note:

1. The data object the keys can be part of the labels of the adjacent editable object so in this example `name` can be used as a key rather than `"Customer name"`.
2. An `idStr` or `ObjectIdentifier` can be used instead of a label (in the example `ctl00_MainContent_fmwOrder_ddlProduct` is used instead of the Product label).
3. `withSetter` can be used to assign a different setter to for an object than the standard framework setter. In the above example the City field is set using `Keys` rather than the default behaviour of directly setting the underlying value. This is a common use case for `withSetter` as sometimes there is JavaScript events on a page that will only fire with a key press event. The script attached to this event would not be triggered when the underlying value is set directly but would be when Key press events are sent to the field by using the `Keys` function.
4. Execution of this function can be sped up by providing a parent object to reduce the search scope (in the final line of the example: `setForm(container, data)`).

Example Changing the Setter for all Fields

```
function doKeysOnTextBox(obj, val){
 if (obj.ObjectType === 'Textbox') {
   obj.Keys(val);
 }
 else {
  set(obj, val);
 }
}

function setFormEndPoint() {
  var data =  {
    ctl00_MainContent_fmwOrder_ddlProduct: 'ScreenSaver',
    Quantity: 1,
    'Price per unit:': 30,
    Discount: 15,
    name: 'Julie Smith',
    Street: 'MyStreet',
    City: withSetter('Melbourne', doKeys),
    State:'Vic',
    Zip: 3126,
    Visa: true,
    'Card Nr:': '1234567890',
    'Expire date': '15/07'
  }

  setForm(data, doKeysOnTextBox);
}
```

The above code would fill the form using key presses for TextBoxes and other types of objects using the framework's default `set` *function*

setFormR

`setFormR` speeds up the creating `setForm` scripts. The use of this function is best illustrated by way of example:

Step 1 - Create a Sample Input Data Object (params) and a Call to `setFormR`

```
function setFormRendPoint(){
 var params =  {
    product: 'ScreenSaver',
    quantity: 1,
    priceperUnit: 30,
    discount: 15,
    name: 'Julie Smith',
    street: 'MyStreet',
    city: 'Melbourne',
    state: 'Vic',
    zip: 3126,
    visa: true,
    cardNo: '1234567890',
    expiry: '15/07'
  };
  setFormR(params);
}
```

This is simply a mapping between the internal property name and values

Step 2 - Create the SetForm Template in the Windows Clipboard

Run the endPoint with the target web page open. A `setForm` template will have been created in the windows clipboard. Paste the clipboard content in place of the `setFormR` command.

```
function setFormRendPoint(){
 var params =  {
    product: 'ScreenSaver',
    quantity: 1,
    priceperUnit: 30,
    discount: 15,
    name: 'Julie Smith',
    street: 'MyStreet',
    city: 'Melbourne',
    state: 'Vic',
    zip: 3126,
    visa: true,
    cardNo: '1234567890',
    expiry: '15/07'
  }
 /*
                  product
                  quantity
                  priceperUnit
                  discount
                  name
                  street
                  city
                  state
                  zip
                  visa
                  cardNo
                  expiry
      */
setForm(
        {
         ctl00_MainContent_fmwOrder_ddlProduct: params.,
         "Product:": params.,
         ctl00_MainContent_fmwOrder_txtQuantity: params.,
         "Quantity:": params.,
         ctl00_MainContent_fmwOrder_txtUnitPrice: params.,
         "Price per unit:": params.,
         ctl00_MainContent_fmwOrder_txtDiscount: params.,
         "Discount:": params.,
         ctl00_MainContent_fmwOrder_txtTotal: params.,
         "Total:": params.,
         ctl00_MainContent_fmwOrder_txtName: params.,
         "Customer name:": params.,
         ctl00_MainContent_fmwOrder_TextBox2: params.,
         "Street:": params.,
         ctl00_MainContent_fmwOrder_TextBox3: params.,
         "City:": params.,
         ctl00_MainContent_fmwOrder_TextBox4: params.,
         "State:": params.,
         ctl00_MainContent_fmwOrder_TextBox5: params.,
         "Zip:": params.,
         ctl00_MainContent_fmwOrder_cardList_0: params.,
         "Visa": params.,
```

```
        ctl00_MainContent_fmwOrder_cardList_1: params.,
        "MasterCard": params.,
        ctl00_MainContent_fmwOrder_cardList_2: params.,
        "American Express": params.,
        ctl00_MainContent_fmwOrder_TextBox6: params.,
        "Card Nr:": params.,
        ctl00_MainContent_fmwOrder_TextBox1: params.
        "Expire date (mm/yy):": params.
      }
    );
}
```

The `setForm` *template includes the idStr and the name of the nearest field object for each editable object encountered.*

Step 3 - Clean Up

At this point the user needs to:

1. Delete any properties they don't want to set
2. For each property delete one of the label name or the idStr key value pair (the `<Shift><Delete>` line deletion shortcut is your friend here)
3. Clean up any property names that are labels (such as removing the colon and inverted commas around single word names)
4. Drag the mapped property names (in the comment above) down to become a property of `params.` in the corresponding object value of `setForm` argument

```
function setFormTheForm(){
  var params =  {
                  product: 'ScreenSaver',
                  quantity: 1,
                  priceperUnit: 30,
                  discount: 15,
                  name: 'Julie Smith',
                  street: 'MyStreet',
                  city: 'Melbourne',
                  state: 'Vic',
                  zip: 3126,
                  visa: true,
                  cardNo: '1234567890',
                  expiry: '15/07'
                };

    setForm(
            {
             ctl00_MainContent_fmwOrder_ddlProduct: params.product,
             ctl00_MainContent_fmwOrder_txtQuantity: params.quantity,
             ctl00_MainContent_fmwOrder_txtUnitPrice: params.priceperUnit,
             ctl00_MainContent_fmwOrder_txtDiscount: params.discount,
             ctl00_MainContent_fmwOrder_txtName: params.name,
             Street: params.street,
             City: params.city,
             State: params.state,
             Zip: params.zip,
             Visa: true,
             "Card Nr": params.cardNo,
             "Expire date (mm/yy)": params.expiry
            }
          );
}
```

The above is how the function might look after clean up.

Note that in some cases (such as Product and Quantity) `idStr` was chosen as the means to identify the editable object and in others (such as City and State) the nearest label was chosen. A good guideline to follow is that if an id has clearly been assigned by the web page author and descriptively named e.g. (`..ddlProduct and ..txtQuantity`) then use it. This id is likely to be stable and will be most reliably found by TestComplete. If however the `id` has been auto assigned (such as `..TextBox3` for City and `..TextBox4` for State) then it is best to use the label text of the adjacent label. The `setForm` statement will be more readable and if another developer replaces the auto-generated id in the web page with a descriptive name the `setForm` function will still work as it is referencing the label not the id.

HTML Grid Utils

TestComplete has some out of the box features for dealing with grids but these can be tricky as they return objects that are different representations of the UI than you would see if using the object spy or `seek` functions described above. The Zenith Framework has some useful utilities for dealing with HTML grids in the `HtmlGridUtils` unit.

The main functions are:

- `eachCellSimple`
- `eachCell`
- `cell`
- `readCell`
- `readGrid`
- `setGrid`

These functions are described below and the examples provided are based on the SmartBear sample web site List of All Orders page:

	Name	Product	#	Date	Street	City	State	Zip	Card	Card Number	Exp	
☐	Paul Brown	ScreenSaver	2	03/12/2010	5, Ringer Street	Las Vegas, NV	US	748	MasterCard	123456789012	02/07	
☐	Mark Smith	FamilyAlbum	1	03/07/2010	9, Maple Valley	Whitestone, British	Canada	76743	VISA	770077007700	01/09	
☐	Steve Johns	ScreenSaver	1	02/26/2010	17, Park Avenue	Salmon Island	Canada	21233	MasterCard	444555444555	03/09	
☐	Charles Dodgeson	MyMoney	1	02/15/2010	45, Stone st.	Bringtone, TX	US	23233	American Express	333222333222	07/10	
☐	Susan McLaren	MyMoney	1	01/05/2010	7, Flower Street	Earlcastle	Great Britain	21444	MasterCard	999888777888	04/10	
☐	Bob Feather	FamilyAlbum	1	12/31/2009	14, North av.	Milltown, WI	US	81734	VISA	111222111222	06/08	
☐	Samuel Clemens	MyMoney	2	12/21/2009	3, Garden st.	Hillsberry, UT	US	53665	MasterCard	398743242342	03/09	
☐	Clare Jefferson	FamilyAlbum	2	12/04/2009	23, Owk Street	Greentown, CA	US	63325	MasterCard	770000770000	03/08	

eachCellSimple

`eachCellSimple` is the most basic grid function. It enables the sequential processing of each cell in a grid and makes no assumptions about the structure of the grid. This is a useful fall back function to use with non-standard grids.

The parameters are as follows:

`grid - required:`
An HTML grid

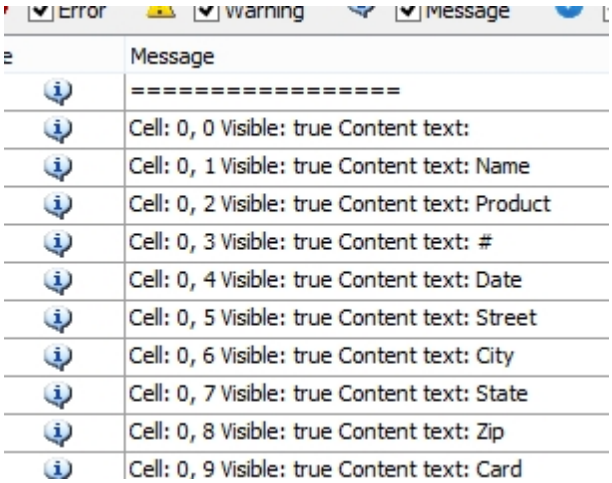
`rowCellFunction - required:`
A function of the form: `function cellTest(cell, rowIndex, colIndex, arFullRow)` that does not return a value. This function will be executed on each cell of the grid

`rowSkipCount - optional [default: 0]`
The number of rows to skip (can be used to skip title row

Example

```
function eachCellSimpleEndPoint() {

  function cellTest(cell, rowIndex, colIndex, arFullRow){
      if (colIndex === 0){
        log('=================')
      }
      log('Cell: ' + rowIndex + ', ' +  colIndex + ' Visible: ' + (cell.Height
> 0)
      + ' Content text: ' + cell.ContentText);
   }

  var grid = seekInPage({IdStr: 'ctl00_MainContent_orderGrid'});
  eachCellSimple(grid, cellTest);
}
```

The above logs out the content of each cell of the "All Orders" grid. The result is as follows:



This fragment shows the title row - the content of the whole grid is logged in this way

Note that although not used in this example the arFullRow parameter can be useful in enabling row by row processing of the grid. During execution this argument is populated with all the cells of the current row.

eachCell

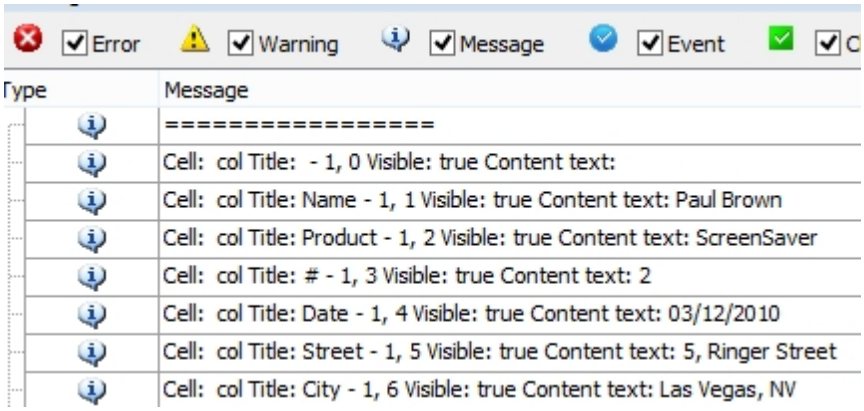
`eachCell` is similar to `eachCellSimple` but for the following differences:

1. This function relies on the first row of the grid being the title row
2. The `rowCellFunction` includes a columnTitle parameter: `function cellTest(cell, colTitle, rowIndex, colIndex, arFullRow)`
3. The `rowSkipCount` parameter defaults to 1 (not 0) which means the title row is skipped by default

Example

```
function eachCellEndPoint() {

  function cellTest(cell, colTitle, rowIndex, colIndex, arFullRow){
    if (colIndex === 0){
      log('=================')
    }
    log('Cell: ' + ' col Title: ' + colTitle + ' - ' + rowIndex + ', '
        +  colIndex + ' Visible: ' + (cell.Height > 0) + ' Content text: '
        + cell.ContentText);
  }

  var grid = seekInPage({IdStr: 'ctl00_MainContent_orderGrid'});
  eachCell(grid, cellTest);
}
```

The above logs out the content of each cell of the "All Orders" grid. Note the title is now available within the `cellTest` *function The result is as follows:*



Shown above is the first record logged (the title row has been skipped)

cell

Returns the first cell in a standard grid matching criteria provided:

The parameters are as follows:

`grid - required:`
An HTML grid

`headerArray - required:`
An array of header column names including the lookup fields and the single return field. Lookup fields should be prefixed by a tilde

`valueArray - required:`
An array of header column values to look up. These must be in the same order as the header array (there should be one less element in this array as there will not be an element for the return value field)

Example

```
function cellEndPoint() {
  var grid = seekInPage({IdStr: 'ctl00_MainContent_orderGrid'});
  var targetCell = cell(
                      grid, ['~City',           'Name'],
                            ['Salmon Island']
                    );
  log(targetCell.contentText);
}
```

The above code locates the first cell with a City field matching "Salmon Island" and returns the Name cell of the same row: "Steve Johns" is logged. Note: If no matching cell was found a stub object would be returned with a single `Exists` *property that with the value of* `false`.

readCell

Calling `readCell` is effectively the same as calling `cell` and applying a `read` function to the result. The first three parameters are the same as `cell` and an optional `readFunction` parameter has been added. The default for this parameter is the framework's default `read` function.

Example

```
function readCellEndPoint() {
  var table = seekInPage({IdStr: 'ctl00_MainContent_orderGrid'});
  var name = readCell(table, ['~City',         'Name'],
                             ['Salmon Island']);
  checkEqual('Steve Johns', name);
}
```

The above example is very similar to that for `cell` *but there no need to read the* `contentText` *of the result. The framework* `reader` *does this as part of the function.*

readGrid

`readGrid` assumes a standard grid and returns an array of JavaScript objects from the cells of the grid (title rows are skipped). The only parameter is the grid itself.

Example

```
function readGridEndPoint(){
  var grid = seekInPage({IdStr: 'ctl00_MainContent_orderGrid'});
  var content = readGrid(grid);
  toTemp(content);
}
```

The content of the grid is returned. The result is the following (shown in JSON format):

```
[
  {
    "": "",
    "Name": "Paul Brown",
    "Product": "ScreenSaver",
    "#": "2",
    "Date": "03/12/2010",
    "Street": "5, Ringer Street",
    "City": "Las Vegas, NV",
    "State": "US",
    "Zip": "748",
    "Card": "MasterCard",
    "Card Number": "123456789012",
    "Exp": "02/07"
  },... and many more
```

setGrid

Applies the standard Framework set function to fields specified via the arrays passed in. Can set multiple fields on multiple rows in one call:

The parameters are as follows:

`grid - required:`
An HTML grid

`headerArray - required:`
An array of header column names including the lookup fields and the single return field. Lookup fields should be prefixed by a tilde column headers without a tilde will be regarded as setter targets

`valueArray - required:`
An array of header column values to look up or set in the same order as the header array (there should be the same number of elements as the header array)

Example

```
function setGridEndPoint_SetSelectedCol() {
  var grid = seekInPage({IdStr: 'ctl00_MainContent_orderGrid'});
  setGrid(grid, ['~Name'     , '' ],
                ['Steve Johns', true],
                ['Mark Smith',  true]
  );
}
```

The above will set the checkboxes under the untitled column in the grid to true for Steve Johns and Mark Smith

	Name	Product
☐	Paul Brown	ScreenSa
☑	Mark Smith	FamilyAlb
☑	Steve Johns	ScreenSa

A Final Word on GridUtils

All of the above grid functions but for `eachCellSimple` rely on the assumption that the grid is a very standard format (e.g. a single header row and no repeated column names). They are aimed at being simple to use in the majority of cases, not to cover every possible scenario. In more unusual cases the user will need to fall back to using `eachCellSimple`.

Exceptions, Ensure and Checks

There are broadly speaking two types of error conditions that may be encountered in a test, those that terminal, i.e that prevent the test from proceeding sensibly (such as the target website being down) and those that are not, such as an incorrect calculation value. When a terminal condition is encountered the test iteration, test case or test run needs to be aborted. The mechanism for this is the use of Exceptions. When a non-terminal error is encountered then a result needs to be logged the mechanism for this is TestComplete's logging functions. The Zenith Framework provides many convenience functions to simplify working with both these types of errors.

Exceptions

Exceptions interrupt the execution of a function. General information on exceptions in JavaScript can be found on the web, but with the Zenith Framework the user does not need to create their own exceptions. There are number a functions provided to generate exceptions and interrupt control flow of a test run. Also raw exceptions in TestComplete can sometimes result in a incomplete call stack which makes it hard to track down the source of the exception. `throwEx`, the Zenith Framework's exception function, will always log an error prior to throwing the exception. This ensures a complete call stack.

throwEx

Throws an exception and causes the current test iteration, test or test run to be aborted. Note that `throwEx` is rarely used. The same functionality can be achieved by using `ensure` (described in the sections below).

The following is a sample of valid calls to `throwEx`:

```
throwEx('person is null')
throwEx('person is null', 'long message about this error...')
throwEx('target web site down', ABORT_TEST_TOKEN())
throwEx('whole environment down', ABORT_RUN_TOKEN())
throwEx('whole environment down', 'I am going home!!!', ABORT_RUN_TOKEN())
```

The parameters are as follows:

`message - required:`
A message that will be posted to the test log

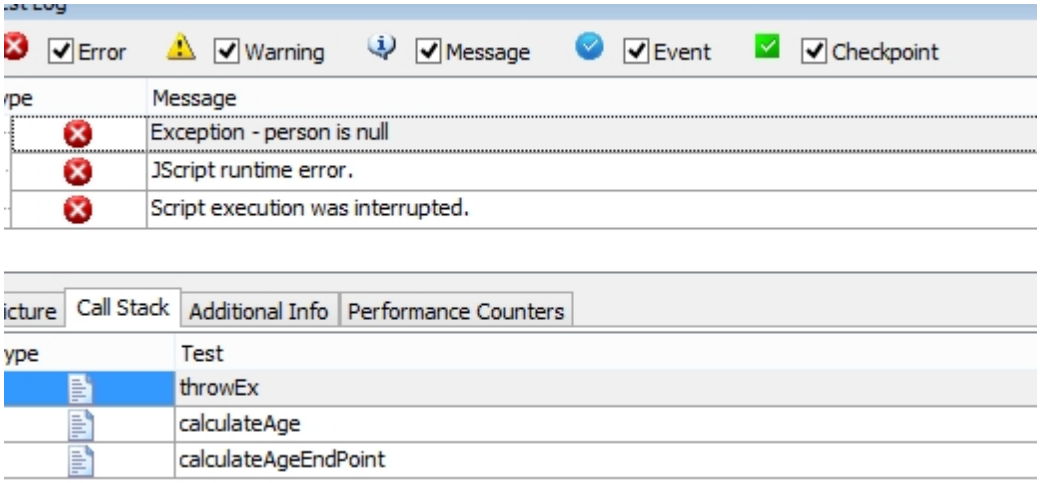
`detail message - optional: [default: message]`
A message that is posted to "Additional Information" in the TestComplete log

`abortLevel - optional [default: ABORT_ITERATION_TOKEN()]`
Controls how the exception affects the test run and can be one of the following values:

- `ABORT_ITERATION_TOKEN()`
- `ABORT_TEST_TOKEN()`
- `ABORT_RUN_TOKEN()`

Example

```
function calculateAge(person){
  if (hasValue(person)){
    return ageCalc(person.dob)
  }
  else {
    throwEx('person is null');
  }
}

function calculateAgeEndPoint() {
  calculateAge(null);
}
```



The above shows a call to `throwEx` *and resulting log call stack. If this exception was triggered in a test run the test would move to the next iteration (reflecting the default* `abortLevel`*)*

Verification - Ensure vs Checks

There are two categories of verification functions in the Zenith Framework. These are **ensure** and **checks**. `ensure` (which is also known as an assertion) and `check*` differ as follows:

1. Ensure aborts the execution of a test iteration if some mandatory condition is not met. For example, if the first step of a test is to read a data file there may be a call to `ensure` the data file exists. In contrast checks will execute some sort of test on the application state and log success if the test passes or log a failure if the test fails. The purpose of using `ensure` is to cause a test script to fail completely and early at the point where a terminal condition is encountered. This makes it easier to fix.
2. If `ensure` fails it will cause an exception to be thrown and the test run will move on to the next test iteration, test or abort the run. Checks return (a Boolean) result to the calling function and the function can continue, optionally changing its control flow based on the result of the test. So a test iteration can have many failed checks but there will only ever be one `ensure` failure.
3. `ensure` does not affect the test run or the test log if the underlying condition is met. Checks will always log something, a "green dot" checkpoint if the test passes or a "red dot" error if the test fails. This changes the nature of informational messages that are used with `ensure` and checks. An `ensure` that a file exists will have a message like "File does not exist: [file path]" a `check` that a file exists will have a message like "Checking file exists: [file path]". The `check` message makes sense equally as part of a success or failure log message, the `ensure` message will only ever be seen in the case of a failure.

ensure

Assertions are implemented in the Zenith Framework via the `ensure` function. This function simply checks a condition and then calls `throwEx` if the condition is false. As such the parameters are the same as for `throwEx` but for a preceding Boolean which represents the condition to be tested.

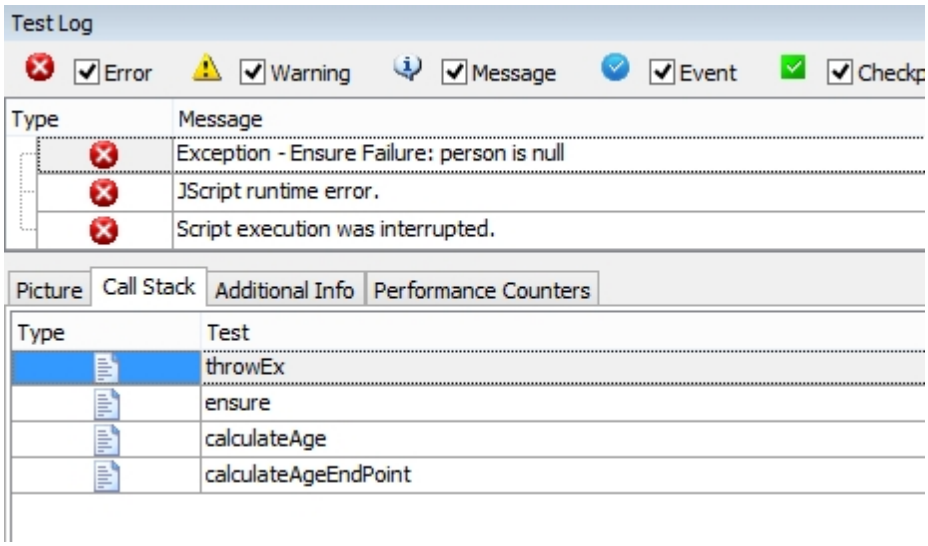
The following is a sample of valid calls to `ensure` (someCondition is a Boolean):

```
  ensure(someCondition)
  ensure(someCondition, 'error message')
  ensure(someCondition, 'error message', 'long message about this error...')
  ensure(someCondition, 'target web site down', ABORT_TEST_TOKEN())
  ensure(someCondition, 'whole environment down', ABORT_RUN_TOKEN())
  ensure(someCondition, 'whole environment down', 'I am going home!!!',
ABORT_RUN_TOKEN())
```

Note: As described above if `someCondition` *was true* `ensure` *would not do anything*

Example

```
function calculateAge(person){
  ensure(hasValue(person), 'person is null');
  return ageCalc(person.dob);
}

function calculateAgeEndPoint() {
  calculateAge(null);
}
```



The above example achieves the same result as `throwEx` with less code

Checks

As described above `check` functions always log and return true or false result without interrupting the test. The base check function is `check`. There are other more specialised check functions but they all perform in a similar way. As such, check will be discussed in some detail while other functions will only be outlined, focussing on what new functionality they bring.

check

Checks a condition logs success or failure. The parameters are as follows:

`condition - required:`
The condition to be tested true / false

`message - optional: [default: empty string]`
A message that follows a prefix "Check" in the main message log

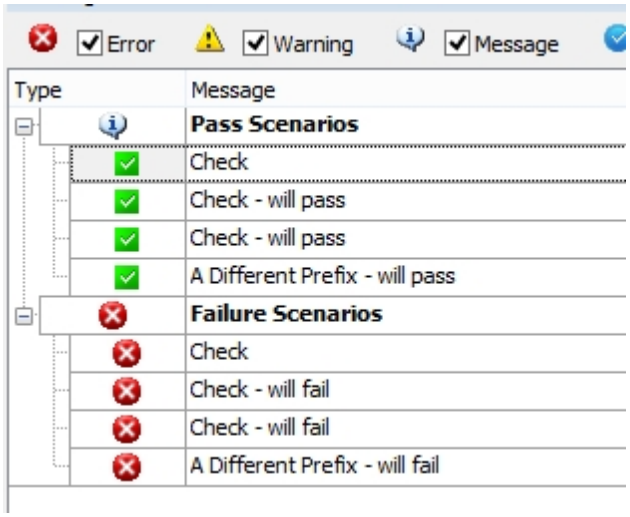
`additionalInfo - optional [default: empty string]`
Is Logged to the Additional Info pane

`prefixOverride - optional [default: "Check"]`
Replaces the prefix in the main log message message

Example

```
function check_UsersGuideEndPoint(){
  pushLogFolder('Pass Scenarios');
  check(true);
  check(true, "will pass");
  check(true, "will pass", "More info ...");
  check(true, "will pass", "More info ...", "A Different Prefix");
  popLogFolder();

  pushLogFolder('Failure Scenarios');
  check(false);
  check(false, "will fail");
  check(false, "will fail", "More info ...");
  check(false, "will fail", "More info ...", "A Different Prefix");
  popLogFolder();
}
```

The above produces the following log output



Note: "More info ..." is logged to the Additional Info panel (not shown)

checkFalse

Performs exactly the same as `check` but checks the condition is false instead of true.

checkEqual

Checks if two objects are equal using the framework's built in `areEqual` function. The following is a sample of valid calls to `checkEqual`:

```
checkEqual(expected, actual);
checkEqual(expected, actual, "test for log message");
```

If the check fails the two objects will be converted to stings and posted to the test log "Additional Info".

checkContains

Checks for a string inside another string. The following is a sample of valid calls to `checkContains`:

```
checkContains(hayStack, needle, infoMessage, caseSensitive);
checkContains("Dog cat", "cat");
checkContains("Dog cat", "cat", "looking for kitty", true);
checkContains("Dog cat", "dog", "looking for rover", false);
```

Note on optional params: `infoMessage` *will be included in the main log message* `caseSensitive` *is false by default*

checkTextContainsFragments

Checks for multiple text fragments within a string. Splits the expected string pattern on * and searches for them in order in the target string. This function also standardises line endings and is case sensitive. The following is a sample of valid calls to `checkTextContainsFragments`:

```
checkTextContainsFragments(pattern, target);
```

Both the pattern string and the target are required parameters

Example

```
function checkTextContainsFragmentsEndPoint(){
  var pattern = 'one *two *Three *' + newLine() + newLine() + '\r' + 'four';
  var target = 'one two Three ' + newLine() + newLine() + '\r' + 'four';
  var result = checkTextContainsFragments(pattern, target);
  check(result);

  var target = 'one Three two  ' + newLine() + newLine() + '\r' + 'four';
  result = checkTextContainsFragments(pattern, target);
  checkFalse(result);
}
```

checkTextContainsFragmentsFromFile

This is a variant of `checkTextContainsFragments` but takes a file name instead of a pattern string. This file is read from the framework TestData directory. In all other ways it is the same as `checkTextContainsFragments`.

Example

```
function checkTextContainsFragmentsFromFileEndPoint(){
  var target = 'one two Three ' + newLine() + newLine() + '\r' + 'four';
  checkTextContainsFragmentsFromFile('textFragments.txt', target);
}
```

checkText

This function checks one whole slab of text against another and logs differences to TestComplete's built in text diff viewer.

```
  checkText(expectedText, actualText)
```

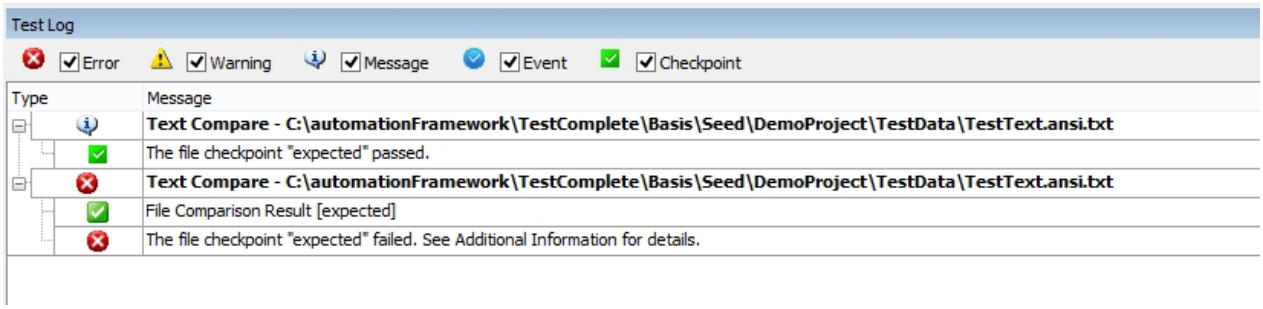
Both the above parameters are required. For an example see `checkTextFromFile`

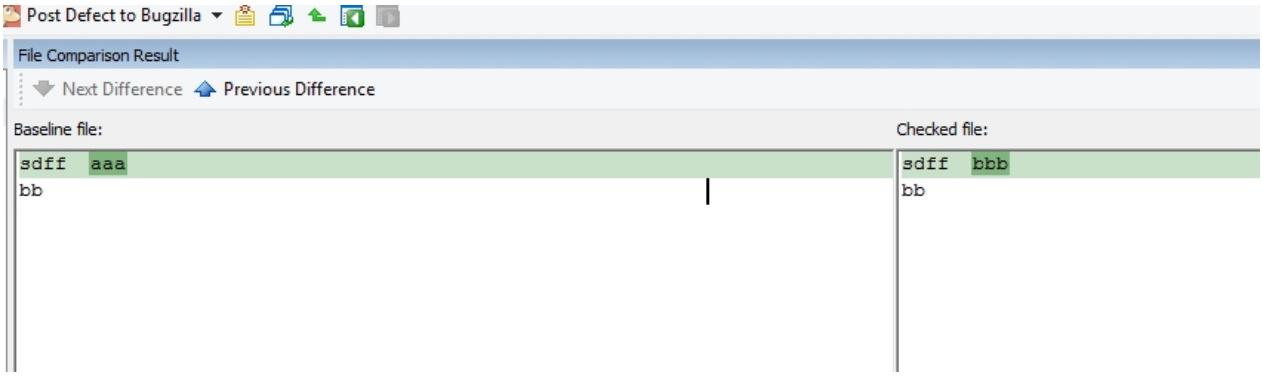
checkTextFromFile

This is a variant of `checkText` but takes a TestData file name instead of a string as the expected value. It also takes an optional file encoding parameter (defaults to **`aqFile.`**`ctANSI`)

```
function checkTextAgainstTextFileEndPoint() {
  var testFile = 'TestText.ansi.txt'
  var actualText = testDataString(testFile);
  // pass
  checkTextAgainstTextFile(testFile, actualText);
  // Fail
  actualText = aqString.Replace(actualText, 'a', 'b');
  checkTextAgainstTextFile(testFile, actualText);
}
```

The above produces the following in the test log. The File Comparison Result is extremely useful when comparing large text files.





Waiting on the AUT

Often throughout an automation the automation will need to wait for the application to be in a particular state before proceeding. This might be waiting for a web page to load, waiting for a button to become enabled or waiting for a processing message to disappear from the screen. Using a fixed Delay method in these situations leads to very slow and or unreliable tests. Rather than an arbitrary delay it is far better to wait for something specific such as a property changing. There a large number of TestComplete Wait methods for this purpose but using the utility functions provided by the Zenith Framework is usually more convenient:

seek

As described in Interacting with the UI all seek functions have a `timeout`. So if the automation needs to wait for an object to become visible or change state (such as become enabled) then just calling a seek function with an adequate time out will be all that is needed. The automation will not delay any longer than it has to. See Interacting with the UI for further details.

```
var userNameTxt = seekByIdStr(30000, 'ctl00_MainContent_username');
```

Waiting for a very slow web page, up to 30 seconds

waitActivePage

`waitActivePage` will wait for up to `timeout` for a web page in the first tab of the browser is "ready", i.e. JavaScript executed and images loaded. It returns the page as an object or a stub object with an Exists property which is false.

```
var loadedPage = waitActivePage(20000);
```

Waits for up to 20 seconds for the page to finish loading

activePage

`activePage` returns the page on the first tab of the active browser. It takes an optional Boolean `waitPage` and `timeout` parameter. If `waitPage` is true then this function behaves exactly the same as `waitActivePage`.

```
var loadedPage = waitPage(true, 20000);
```

Waits for up to 20 seconds for the page to finish loading

waitRetry

`waitRetry` is the most powerful an flexible wait function. The above methods simply wait on a UI property (`seek`) or wait on a web page load (`waitActivePage`, and `activePage`). `waitRetry` is far more flexible, it enables the user to wait for the AUT to be in a particular state and execute a function between checks. The functions described above are all implemented using `waitRetry`. All the below are valid calls to `waitRetry`.

The following is a sample of valid calls to `waitRetry`:

```
  waitRetry(rowExistsInTable)
  waitRetry(rowExistsInTable, "Waiting for record to load")
  waitRetry(rowExistsInTable, 60000)
  waitRetry(rowExistsInTable, clickRefreshButton)
  waitRetry(rowExistsInTable, clickRefreshButton, 60000)
  waitRetry(rowExistsInTable, clickRefreshButton, 60000, 3000)
  waitRetry(rowExistsInTable, clickRefreshButton, 60000, 3000, "Waiting for
record to load")
```

The parameters are as follows:

`isCompleteFunction - required`
A parameterless function that tests if the desired end state has been reached (returns a Boolean)

`retryFuction - optional [default: doNothing]`
A parameterless function that returns nothing but executes some action on the AUT between each check (e.g. clicking a reload button)

`timeoutMs - optional [default: 10000]`
How long to keep checking for

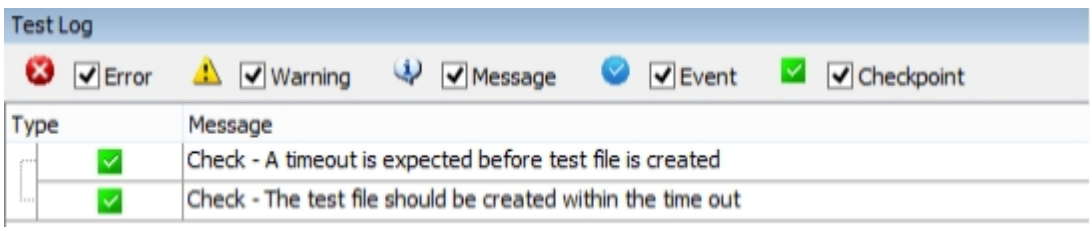
`retryPauseMs - optional [default: 0]`
How long to pause between sets of executing `isCompleteFunction` and `retryFuction`.

`indicatorMessage - optional [default: empty string]`
An optional message that comes up on the TestComplete indicator while the `waitRetry` process is executing

Example

```
function waitRetryUnitTest() {
  var testFile = tempFile('myTestFile.txt');

  function isComplete(){
    return aqFileSystem.Exists(testFile);
  }

  function retry(){
    retryCount ++;
    if (retryCount > 5){
      stringToFile('dsfdf', testFile);
    }
  }
  // the file will not be created within the retry period
  aqFileSystem.DeleteFile(testFile);
  var retryCount = 0;
  var result = waitRetry(isComplete, retry, 4000, 1000, 'timeout expected');
  checkFalse(result, "A timeout is expected before test file is created");

  // the file will be created within the retry period
  aqFileSystem.DeleteFile(testFile);
  retryCount = 0;
  result = waitRetry(isComplete, retry, 10000, 1000, 'Waiting for file -
expect success');
  check(result, "The test file should be created within the time out");
}
```

Running the above code results in the following log



Dealing with Empty Values

One issue that all languages need to deal with and particularly untyped languages is how to deal with "nothing like" values. In JavaScript "nothing like" values of particular interest are: `null` and `undefined`:

- `null` is a value that has been returned from some other function or passed through a parameter as `null`, that is explicitly assigned a missing or unknown value by another part of the application
- `undefined` has simply not been assigned. E.g. a function that takes two parameters is called with one then the second parameter is will `undefined` also referencing a non-existent property on an object will return `undefined`.

Two other forms of "nothing like" values that need to be dealt with in an automation context are empty strings and objects that don't exist:

- **Empty strings** (`""`) may or may not represent a "nothing like value" depending on the context. In cases such as reading text of an empty label on a UI it does but in other cases, such as passing an empty string as a delimiter in a string function it does not.
- **Objects that don't exist:** As described in Interacting With the UI, when a UI function cannot find an object it returns a stub object with a single `Exists` property with the value of `false`. This can also be regarded as a value that represents nothing.

There are two framework functions that can be used to deal with "nothing like values" with slightly different purpose:

- `def` - is used to replace `null` or `undefined` values with a default and is very commonly used to overload functions by provide default parameter values
- `hasValue` - is used to determine if a value is `null` or `undefined` or an empty string or a UI object that does not exist and is used in normal control flow

def

`def` simply takes an initial value and if this value is `null` or `undefined` it will return an alternate default value.

The following is a sample of valid calls to `def`:

```
def(initialValue, defaultVal)
def(initialValue, defaultVal1, defaultVal2, defaultVal3)
```

The parameters are as follows:

`initialValue - required`
The initial value that is to be defaulted if it is `null` or `undefined`

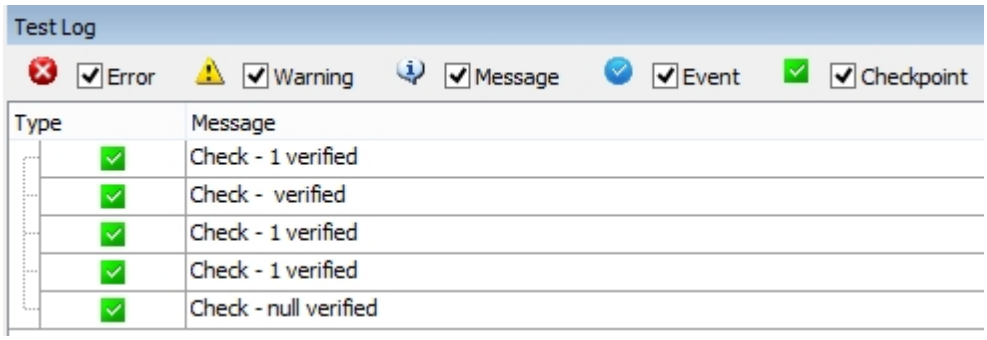
`defaultVal* - required`
The value to default to. Note there can be many default values in which case the default returned will be the first of these values which is not `null` or `undefined`. This can be particularly useful for disambiguating object properties on objects that have come from an external source e.g:

```
var name = def(obj.name, obj.firstName, obj.givenName, obj.givenName,
'No Name Provided');
```

Another way of thinking about `def` is that given a list of arguments it will return the first which is not **`null`** or **`undefined`** or the last if all are **`null`** or **`undefined`** .

Example

```
function defUnitTest(){
  // myVar will be undefined
  var myVar;
  var deffedVar = def(myVar, 1);
  checkEqual(1, deffedVar);

  // empty string is treated as a value and not defaulted
  myVar = "";
  deffedVar = def(myVar, 1);
  checkEqual("", deffedVar);

  // null is defaulted
  myVar = null;
  deffedVar = def(myVar, 1);
  checkEqual(1, deffedVar);

  // the first non-null and non-undefined argument is returned
  myVar = null;
  deffedVar = def(myVar, undefined, 1);
  checkEqual(1, deffedVar);

  // if all arguments are null or undefined will fall back to the last
argument
  myVar = undefined;
  deffedVar = def(myVar, undefined, undefined, null);
  checkEqual(null, deffedVar);
}
```

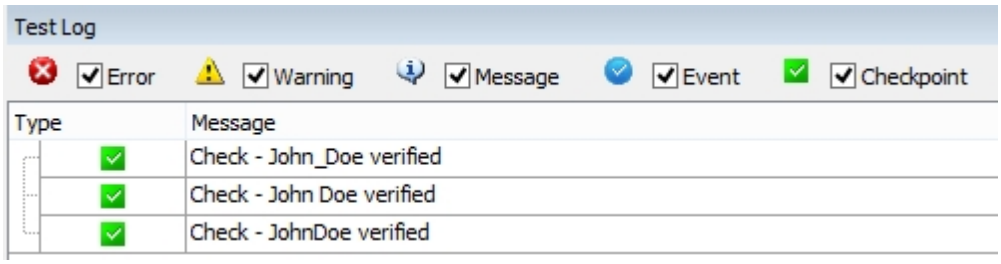


Use Case Example

The most common use case for `def` is in creating overloaded functions

```
function addTwoStrings(str1, str2, /* optional */ delim){
  delim = def(delim, ' ');
  return str1 + delim + str2;
}

function addTwoStringsEndPoint(){
  var result;
  result = addTwoStrings('John', 'Doe', '_');
  checkEqual('John_Doe', result);

  // overloaded: delimiter should default to space
  result = addTwoStrings('John', 'Doe');
  checkEqual('John Doe', result);

  // as an empty string is treated as a value it will not be defaulted
  result = addTwoStrings('John', 'Doe', '');
  checkEqual('JohnDoe', result);
}
```

The use of `def` *on the* `delim` *parameter turns* `addTwoStrings` *into a function that can take two or three parameters.*



hasValue

`hasValue` takes a single object as a parameter and returns true if the object is NOT any of **`null`** or **`undefined`** or an **empty string** or a **UI object that does not exist.** It is most commonly used in general control flow especially when interacting with the UI.

Example

```
function hasValueDemo_EndPoint(){
  var welcomeLabel = seekInPage(0, {
                                     contentText: 'Welcome*',
                                     Visible: 'True'
                                    });

  if (!hasValue(welcomeLabel)){
    logIn();
  }
}
```

The above uses hasValue to determine the existence of a welcome label (which indicates the user is logged in). If the label is not present (i.e. hasValue(label) === false) then a login is performed

Handling Known Defects

expectDefect and endDefect

In the perfect world any defect surfaced by an automated UI test would be fixed the same day it is found but in the real world this does not always happen. If a minor defect is found it might be logged in a defect tracking system and put on a backlog to be fixed at a later date. This can be a problem in the following test runs because time can be wasted reconciling known errors with errors in the test log. The framework solves this problem by using `expectDefect` and `endDefect` to mark the position of an expected error in the log file. At the end of the test run the framework parses log file and generates a report of unexpected errors ignoring those that are marked with expectDefect. This report also highlights sections of a test where errors were expected but have not occurred. This indicates a defect which has been fixed and the defect expectation should be deactivated.

The parameters are as follows:

`defectId - required:`
A message or identifier for the defect expectation - often corresponds to a bug id in a bug tracking system

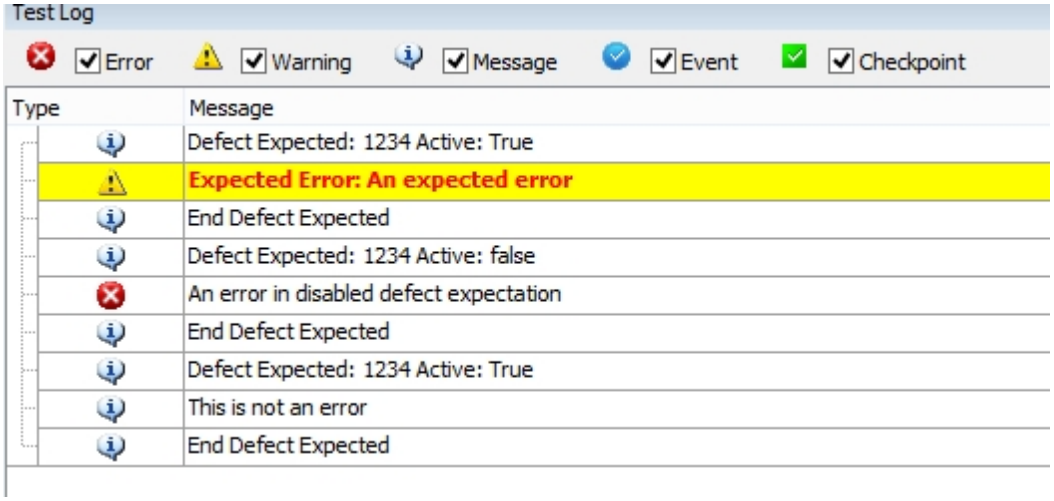
`active - optional: [default: true]`
When a defect has been fixed this parameter can be set to false. If the software regresses then error will be logged as an error but there will be evidence in the automation code the software has been broken at this point previously.

Example

```
function expectDefectUnitTest(){
  // this error would be ignored by the log file parser
  expectDefect(1234);
  logError('An expected error');
  endDefect();

  // the log file parser would log an error because although the error is
preceded by expectDefect
  // the defect expectation as been deactivated
```

```
  expectDefect(1234, false);
  logError('An error in disabled defect expectation');
  endDefect();

  // the log file parser would log an expected error that
  // did not occur as there is an error expectation but no error logged
  expectDefect(1234);
  log('This is not an error');
  endDefect();
}
```

Note the expected error has be turned into a warning in the test log



expect_defect

`expect_defect` performs exactly the same function as `expectDefect` but has been written to be included in a list of `validators` on a testItem:

```
{
      id: 5,
      when: '4',
      then: '44',
      message: 'item 3',
      toDo: 'Need to think of some more validation here',
      validators: [
                    expect_defect("that's gotta hurt", true),
                    demo_validation_will_fail,
                    another_validation
                  ]
 }
```

In the above example even if `demo_validation_will_fail` *logged an error the framework would surround this with a defect expectation downgrade this error to a warning. This is similar to what is shown in the log above. If* `another_validation` *logged an error this would remain an error.* `expect_defect` *only affects the validation that immediately follows it, there is no need to use* `endDefect` *or an equivalent function in the validation list.*

Higher Order Functions

A detailed description of how and whys of higher order functions is well beyond the scope of this user manual. Your consultant will provide detailed training on the use of higher order functions as you become familiar with the framework and test development workflow. The purpose of this section is to provide a brief introduction, demonstrate a couple of common use cases and to provide link to our main higher order function tool: underscore.

What are High Order Functions (HOFs)?

Higher order functions are simply functions that either take other functions as parameters or return a function as a result. In JScript, functions can be passed around and produced as results of functions just like data.

Why High Order Functions?

Higher order functions provide a powerful way to reduce the amount of code you need to write and make code easier to read and less prone to defects than code written without HOFs. Code written using HOFs also tends to be more declarative than code written without, i.e. it says what it does rather than obscuring the intent of the code with implementation details.

Functions that Take Functions as Parameters

When a function takes another function (lets call this the "executive" function) as a parameter this can enable this function to be highly flexible without being complex. Whole behaviours can be injected into the executive function without adding complexity.

If for example a function was to login to the same web page then use one of a number of different navigation strategies to navigate to a results page before validating the results then one way to do this would be to add a data parameter to the function to represent the navigation strategy. Using such an approach, this function would then have to include all the logic for selecting a strategy based on the navigation strategy parameter and the implementation of the navigation strategies themselves. It is far neater to have the common log in and validation code in the main function and pass the navigation code in as a function parameter. The main function can then just login and call the navigation function then execute the validation. A call to this function would then look something like this:

```
validatePage(navigateViaMainMenus)
```

The navigation being passed could then be run and tested in isolation and new navigation strategies could be added by just by passing in a different navigation function.

Another use case for passing a function to another function is to factor out control flow from a function and reuse it. For example here is a code snippet for doing something to each file in a directory:

```
// .. code before this
var iterator = aqFileSystem.FindFiles(folder, searchPattern, recursive);

if (hasValue(iterator)){
 while (iterator.HasNext()){
   var info = iterator.Next();
   // do stuff
 }
}
```

Repeating this code every time we want to process files in a directory is quite messy so this code is put into a single function called `eachFile`

```
function eachFile(folder, fileFolderFunc, searchPattern, recursive){
  var iterator = aqFileSystem.FindFiles(folder, searchPattern, recursive);

  if (hasValue(iterator)){
    while (iterator.HasNext()){
      var info = iterator.Next();
      fileFolderFunc(info.Path);
    }
  }
}
```

Now the messy iteration out of the way all the user needs to think about is what they want to do with the file path and put that in a function this function will be called for each file.

Example Listing All Files

```
function listFilesorFolder(folder, searchPattern, recursive){
  var result = [];
  function addPath(path){
    result.push(path);
  }
  eachFile(folder, searchPattern, recursive, addPath);
  return result;
}
```

`addPath` simply adds the path to the result. This function is passed to eachFile and is called for each file in the folder. The result is a list of all files in the folder.

Functions that Return Functions

Functions that return functions can be considered "pre-configurable functions". Some of the parameters can be added at the declaration and some can get passed in when the function is called.

The most common example of the use of such functions in the test framework is setting creating flexible business rule validators. The following is an example of a typical validator as described in Test Cases and Test Items.

```
function demo_validation(apState, item, runConfig){
  check something here
}

function testItems(runConfig){
  var result = [
    {
      id: 1,
      toDo: 'add some validations to this',
      message: 'item 0',
      validators: demo_validation
    }, ...
```

Now imagine there is a test that returns a web service response and we wanted to write a number of test iterations verify a number of different error codes given different inputs. Each iteration would need a slightly different validator to verify a different code. One solution would be to write a large number of similar functions and use these as the validator for the appropriate iteration:

```
function check_for_error_code_45(apState, item, runConfig){
  ....
}

function check_for_error_code_55(apState, item, runConfig){
  ....
}

function check_for_error_code_66(apState, item, runConfig){
  ....
}

...
```

This would result in a lot of code repetition. Another solution would be to add a parameter to the test item.

```
function check_error_code(apState, item, runConfig){
  checkEquals(item.expectedError, apState.errorCode)
}

function testItems(runConfig){
  var result = [
    {
      id: 1,
      toDo: 'add some validations to this',
      message: 'item 0',
      expectedError: 45,
      validators: check_error_code
    }, ...
```

This is a better solution than the first but these items could have multiple checks like this, some could check for a number of error codes others could check a completely different aspect of the application state and these other aspects may have some similarly variable elements as well. In such cases the test item objects can become very large with many and variable properties. These properties will include inputs for interactor and inputs for the validators. This structure can quickly get out of hand and it can get difficult to understand how an item will be execute by looking at it because all the items are so different. The best solution is to move the configurable part of the validator into a higher order function such as shown below:

```
function check_error_code(errorCode){
 function errorCheck(apState, item, runConfig){
  checkEquals(errorCode, apState.errorCode)
 }
 return errorCheck
}

function testItems(runConfig){
  var result = [
    {
      id: 1,
      toDo: 'add some validations to this',
      message: 'item 0',
      validators: check_error_code(45)
    }, ...
```

Now `check_error_code` takes an errorCode parameter and returns a function configured to check the errorCode passed in. The test item now maintains its readability as it is not polluted by validator parameters.

Note if we wanted to test `check_error_code` in isolation we could do as follows:

```
function check_error_codeEndPoint(){
 ... generate mock apState, item, runConfig

 // Should pass
 apState.errorCode = 55;
 check_error_code(55)(apState, item, runConfig);

 // Should fail
 apState.errorCode = 56;
 check_error_code(55)(apState, item, runConfig);
}
```

The call to `check_error_code` returns a function that has been configured to expect an error code of 55. The resulting function is then invoked immediately with the parameters: `apState, item, runConfig`.

The Main HOFs Library Used in the Framework

The core library for using higher order functions is called underscore and can be used in scripts by adding: `//USEUNIT _` to the uses clause at the top of the file. The api documentation is here: underscorejs.org. It pays to go through the examples. There are many useful functions in underscore but `map, reduce, filter, pick, pluck, every` and `chain` are some of the most useful.

Your consultant will help improve your underscore skills as you get opportunities you use it. Underscore will make your automation code much more understandable and maintainable.

Data Driven Testing Using Excel

Why You Should Not Use Excel for Data Driven Tests

Excel driven tests are not the recommended way of passing data to a test in most circumstances. The default method of using a hard coded array of JavaScript objects is simpler to manage and more powerful. Using JavaScript objects enables the user to pass functions into a test as well as data which makes such tests far more flexible. Using a default data record with JavaScript objects also enables a default object to be defined which means that each test item only contains the differences to the default. This makes the intent of the data easier to understand and the test easier to maintain.

If You Must

The only time where using an Excel spreadsheet for test data might be useful is if this data is coming from another source, such as a business analyst and / or there is a very large amount of data for which test validation is relatively uniform and simple. E.g. hundreds of records of income related inputs each with a single expected result for tax return calculation.

The way the Zenith Framework handles excel spreadsheets is to convert this data into an array of JavaScript objects. The function used for this is `worksheetToArray` and is found in the `ExcelParamsLoader` script.

worksheetToArray

The `worksheetToArray` function converts a spreadsheet that complies with certain conventions to an

array of JavaScript objects that can be returned from a `testItems` function.

The parameters are as follows:

`excelFileNameNoPath - required:`
This is the name of the source file to be used which must be saved in the TestData directory. This file must conform to the conventions described in the next section.

`validators - optional [default: null]:`
This is a single validation function or array of validation functions that will be assigned to every item returned. This / these function(s) implement the business rule validation for the test item. See Test Cases and Test Items for more details on how business rule validation is implemented in tests. If no validator(s) are assigned the resulting testItems will have no business rule validation.

Example

The following spreadsheet is in the TestData directory (`DataClassGenTest.xlsx`)

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	id	fieldText	fieldDate	fieldFloat	fieldMixed	fieldInt	fieldCurrency	child:example	fieldTextChild	fieldDate	<Comment>	fieldFloatChild	fieldMixedChild	fieldIntChild	fieldCurrencyChild
2	1	aaaa	1/01/2000	1.23	1/01/2000	1	2.90		aaaaw	1/04/2000		1.23	1/01/2000	1	1.90
3	2	bbbb	2/01/2000	1.4	1.4	2	1.34		bbbbw	2/04/2000		1.4	1.4	2	1.34
4									aaaarr	1/05/2000		1.23	1/01/2000	1	1.90
5									aaaarr	2/05/2000		2.23	2/01/2000	1	1.90
6									aaaarr	3/05/2000		3.23	3/01/2000	1	1.90

This function call below converts this worksheet to an array of JavaScript objects

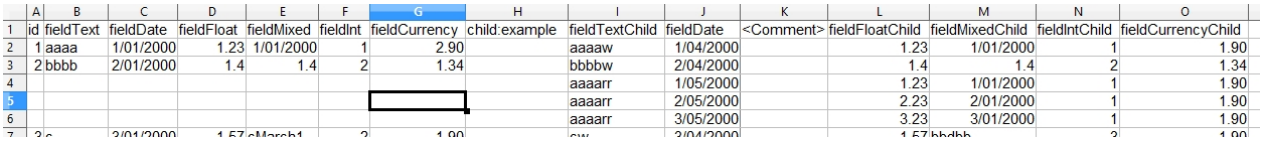
```
function worksheetToArrayEndPoint(){
  function say_hi(obj){
    log('HI ' + obj.id);
  }
  var wsData = worksheetToArray('DataClassGenTest.xlsx', sayHi);
}
```

The content of `wsData` would include the following for id 2:

```
{
    id: 2,
    fieldText: 'bbbb',
    fieldDate: 2000-01-02,
    fieldFloat: 1.4,
    fieldMixed: 1.4,
    fieldInt: 2,
    fieldCurrency: 1.34,
    example: [
               {
                 fieldTextChild: 'bbbbw',
                 fieldDate: 2000-04-02,
                 fieldFloatChild: 1.4,
                 fieldMixedChild: 1.4,
                 fieldIntChild: 2,
                 fieldCurrencyChild: 1.9
               },
               {
                 fieldTextChild: 'aaaarr',
                 fieldDate: 2000-05-01,
                 fieldFloatChild: 1.23,
                 fieldMixedChild: 2000-01-01,
                 fieldIntChild: 1,
                 fieldCurrencyChild: 1.9
               },
               {
                 fieldTextChild: 'aaaarr',
                 fieldDate: 2000-05-02,
                 fieldFloatChild: 2.23,
                 fieldMixedChild: 2000-01-02,
                 fieldIntChild: 1,
                 fieldCurrencyChild: 1.9
               },
               {
                 fieldTextChild: 'aaaarr',
                 fieldDate: 2000-05-03,
                 fieldFloatChild: 3.23,
                 fieldMixedChild: 2000-01-03,
                 fieldIntChild: 1,
                 fieldCurrencyChild: 1.9
               }
             ],
  validators: say_hi
}
```

Note the data has been loaded and the say_hi function has been assigned to the validators property.

Excel Conventions



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	id	fieldText	fieldDate	fieldFloat	fieldMixed	fieldInt	fieldCurrency	child:example	fieldTextChild	fieldDate	<Comment>	fieldFloatChild	fieldMixedChild	fieldIntChild	fieldCurrencyChild
2	1	aaaa	1/01/2000	1.23	1/01/2000	1	2.90		aaaaw	1/04/2000		1.23	1/01/2000	1	1.90
3	2	bbbb	2/01/2000	1.4	1.4	2	1.34		bbbbw	2/04/2000		1.4	1.4	2	1.34
4									aaaarr	1/05/2000		1.23	1/01/2000	1	1.90
5									aaaarr	2/05/2000		2.23	2/01/2000	1	1.90
6									aaaarr	3/05/2000		3.23	3/01/2000	1	1.90

The above is a repeat of the excel worksheet screenshot to make it easy to see the application of the required conventions.

In order for the `worksheetToArray` function to work correctly the following conventions must be followed:

1. Spreadsheets will be stored in the TestData sub directory and will be *.xlsx format
2. Spreadsheets will be named in Pascal case with no spaces or hyphens (e.g. `DataClassGenTest.xlsx`)
3. There will only be one tab per spreadsheet and it with the name left as `Sheet1`
4. The first column in the spreadsheet will be reserved for the test item id which must be an integer and unique within the worksheet
5. All fields other than comment fields and child declarations will be named with a camelCase format (e.g. `incomeTax`)
6. Fields named <Comment> are ignored by the framework
7. A single nested relationship can be represented by putting a single `|child: [child class name]|` column in the column header (optional) e.g. In the above column H: `child:example` will cause an array of child objects to be created and assigned to a property called `example`
8. When there is more than one child record then all the main record fields will be left blank for the child rows after the first child record. In the above example for record id 2 all the header record fields are left blank in rows 4, 5 and 6.

A Final Word on Using Excel Spreadsheets

Be careful of mixed data. Excel uses the first ~ 6 rows to determine the data type of a column. If in a later row you insert a data of a different type then it will returned to the framework as a null value. E.g. If there is a productCode column and the first 8 rows are integers then the tenth row is a string then the record in row 10 will be returned with a null product code. The work around for this is to order the records differently or place a dummy record at the top of the worksheet to force mixed data into the first few rows of the workbook for this field. If using a dummy record this can be removed in code after `worksheetToArray` is called.

Naming Conventions

This section summarises the key naming conventions in the Zenith Framework most of which have been covered in other sections.

Script Naming Conventions

Test Case Scripts Must end in `_Test`. The descriptive part of the test name should also use a name space like convention to divide up test cases into logical sections. See Test Development Workflow for details.

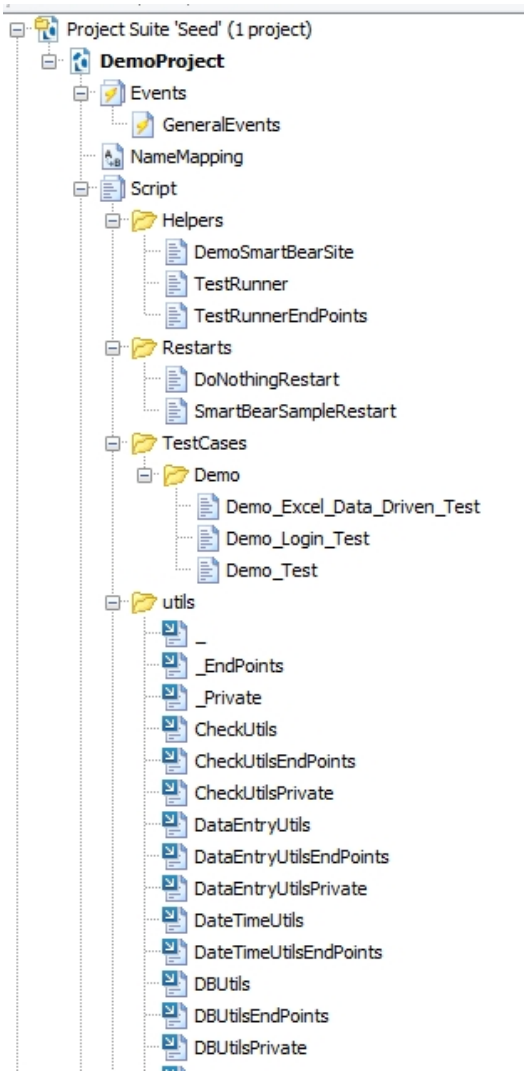
Restart Scripts Must end in `Restart`

Helpers Can be named as to their domain (e.g. `Inventory`)

Utils Must end in `Utils`

EndPoint Scripts Must end in `EndPoints`

Private Scripts Must end in `Private`



More details relating to naming conventions and the purpose of these scripts are available in Framework Structure

Function Naming

Details regarding naming of functions in a test script are available in Test Cases and Test Items. Details regarding the naming conventions around `unitTests` and `endPoints` are available in UnitTests and EndPoints. Other important naming conventions are as follows.

Test Run Function Names

Must end in `TestRun`

```
function demoTestRun(){
  run_amcmeCorp_Tests(
                              {
                                name: 'demo Run',
                                demo: true,
                                mocking: false
                              }
                            );
}
```

Normal Function Names

Must be camelCase e.g. `function myCoolFunction()`

Validators and Test Filters

Must be snake_case e.g. `function demo_validation(apState, item, runConfig)`
`function is_enabled(testName, testConfig, runConfig)`

Object Properties

Must be camelCase e.g.

```
                    {
                      id: 2,
                      when: '1',
                      then: '11',
                      message: 'item 1',
                      givenName: 'Jill',
                      validators: demo_validation
                    }
```

Functions that Return Functions

Must conform to `make + <Return Function Description> + Function` pattern. e.g. `function makeErrorCheckFunction(errorId)`
Note: this rule does not apply to functions that generate validators or test filters these should snakeCase as described above for Validators and Test Filters

Scripting Recommended Practices

For an automation to be successful the test must be run, reconciled and maintained and for this process to be successful the code base of the automation must be maintainable.

General Attributes of Maintainable Code

The practices listed in this section are not arbitrary choices. All these practices are aimed at achieving a script base that has the interrelated characteristics of being small, understandable and testable. If the script

base has these characteristics it will, as a consequence, be maintainable. If the code base is understandable, this will lead to higher levels of reuse, hence the code base will be smaller. A smaller code base will be less prone to defects and will be easier to test. Shared code will also be exercised more frequently so defects will be surfaced and fixed quickly in a smaller number of places. Code that is understandable is also, in itself, easier to test because it is easier to understand the expected behaviour.

Keeping the Code Base Small

Ensure You Understand and Follow All Conventions

The conventions the test framework has been built around enable it to handle a large amount of common test automation functionality and the full use of these conventions will greatly reduce the amount of code you need to write. These conventions are for your benefit. Ensure you understand and use them.

DRY / Look First: Reuse or Refactor Before You Rewrite

DRY = Don't Repeat Yourself and, for that matter, you should also not repeat anyone else. So before you write any function you need to think and Look First within the project for the same or a similar function <Find><Search Project> is your friend. If you find a function that does exactly what you need to do you should just call it. If you find something that does something similar you should refactor the shared functionality out to a separate function and call the common code from both. You may also need to change the name of the old function and edit all the calling scripts where the old name is not or is no longer accurate. Make sure you haven't broken the existing call to the function after you have refactored / moved it - the TestComplete search functionality. **You should never copy and paste an existing function.** Always call the old one or refactor as above.

Don't forget to check the TestComplete help if you need to perform utility functions such as date arithmetic.

Put Code in the Right Helper

Put functions in the helper that makes the most sense for the function it provides and then using //USEUNIT where you call it from.

Making the Code Base Understandable

Clean Up as You Go

All the following recommendations should be applied progressively as you develop a function not just at the end in a big "clean up" just before you have finished a test case.

Test Cases MUST be Independent

All test cases must start from what is deemed to be the Home State e.g. logged in to the main navigation screen / page of the application and must end there or end in a state where the restart script can navigate back to this state. They must be able to be run independently and should NEVER depend on another test case being run first. Rolling over and navigating to the Home State should be handled by the a restart script, not the test.

An Automated Test Case Must Be Focused, Execute Tests Consistent with its Name an When / Then Statement

A test case must accurately reflect the test case as outlined in the source test case. The automation may execute a more detailed set of steps than outlined in source test case but it must not skip steps in the automation (such as validation).

Automated test cases should be focussed. For example if the objective of a test case is to check a calculation it should not check screen validations. Such tests should be split prior to automation.

When deciding how to create and split test cases remember that test cases could fail on the last line in an overnight run. Long rambling test cases are extremely difficult and time consuming to diagnose and maintain in such circumstances. Test case scripts should also be named descriptively and conform to the framework's conventions.

Do Not Use Global Variables

Never use global variables. They create a near impossible debugging / testing experience for anyone using your code. To reproduce or diagnose problems the user would have to know exactly what global variables need to be set and there will be no hint of this by looking at the function. To make a function understandable you must pass all the required information into the function via properly named parameters.

Compose Test Cases of Smaller Understandable Functions

When writing a test case or helper function build the function gradually from sub functions testing and refactoring each function as required. When the function is finished the sub-functions should read like a set of simple instructions as below.

```
function setRereadItemPrices(){
  navigateToSetItemPrices();
  populateSetItemPrices();
  clickOKtoSetItemPrices();

  navigateToItemsList();
  var result = getItemListSellPrice();
  closeItemsList();
  return result;
}
```

Do not start by generating a huge slab of code then refactoring it into sub-functions when you are finished - keep the code broken down as you go. This makes it easier to test each sub-function in isolation.

Keep Functions Short

Try to keep functions, including test cases to less than 40 lines of code. Any longer and you should probably consider splitting the function. If you have to scroll to read a function it is definitely too long and should be split.

Limit Each function to a Maximum of Three or Four Parameters

Functions with few parameters are easier to read and understand than functions with a lot of parameters. Use param objects when you are dealing with a function that needs a lot of data rather than having many individual parameters.

Name functions, Parameters and Variables Carefully and Ensure Naming Conforms to Our Conventions

With the correct variable, parameter and function names your functions will read like a simplified form of English. Conversely poor function naming obscures the intent of a function. Importantly avoid the use of abbreviations - It might be clear to you that soh means stockOnHand when you are writing an inventory test but it will be a mystery to someone else when they have to change your script 6 months later.

Keep Your Privates Private

By default functions in script units other than TestCases and EndPoint units should be declared in private script units unless they are intended to be shared with units outside that script. Test case script units are always effectively private by virtue of Code Sharing / Script Dependency Guidelines so this rule need not be applied in the case of functions within test case script units.

Declare and Assign Variables as Close as Possible to Where they are First Used

Following this practice makes it much easier to refactor, hence enhance the understandability of code later. You can cut and paste code and move code around without breaking it. Conversely if all your variables are declared at the top of a procedure then moving a small slice of code near the bottom of the procedure will result in an error unless you hunt down the variable declaration and initialisation first.

Set Optional Parameters at the Top of Functions

Use the `def()` function to set optional parameters right near the top of functions. It makes the code self documenting.

Avoid Deeply Nested If and / or Select Statements

Code like this is a mess and is really hard to follow - use sub functions and higher order functions to break

up code like this.

```
function deep(a,b,c,d,e){
  if(a) {
    if{b}{
      if (d) {
        if (!e) {
          ...
        }
      }
    }
    else if (c){

    }
  }
}
```

Do Not Use Delay()

It is bad practice to use the `Delay()` function. Instead use one of the functions in the framework or inbuilt TestComplete Wait functions.

if / else Statements – Positive First

Setting out If / else statements based on a positive value first / else negative makes code easier to read - hence easier not to break.

```
function right(){
 if (itsAGirl) {
   // it's a girl
 }
 else {
   // it must be a boy
 }
}

function wrong(){
 if (!itsAGirl) {
   // it must be a boy
 }
 else {
   // it's a girl
 }
}
```

Add Concise Relevant Comments When Checking in To Version Control

The version control history needs to be understandable as well as your scripts. Check in comments such as "Updated script" are not useful.

Making the Code Base Testable

All Test Cases Should Include a Standard TestCaseEndPoint

See TestCaseEndPoints for details

Split Navigation from Non-Navigation Functions

You should also keep non-navigation steps within a single function. If, for example, you have a function which clicks on a button to open a screen, reads the screen then clicks on cancel this function should be three lines that perform the following:

```
clickOpenButton();
var result = readScreen();
closeScreen();
```

By breaking the navigation away from the screen interaction (the read) this enables the readScreen to be tested repetitively in isolation.

All Parameterised Functions Must Have UnitTests or EndPoints

You cannot invoke TestComplete on a sub-function if the function has parameters so all parametrised sub-functions must have a unit test or function immediately below the function itself or in an EndPoint Script file.

Unit Tests Should Perform Required Set Up and EndPoints be Commented With Set Up Requirements

If an endpoint requires (non-obvious) set-up there should be a brief comment on top of the function. Otherwise the unit test should perform the required set up itself: E.g. `//Requires prepare time billing invoice screen to be loaded.`

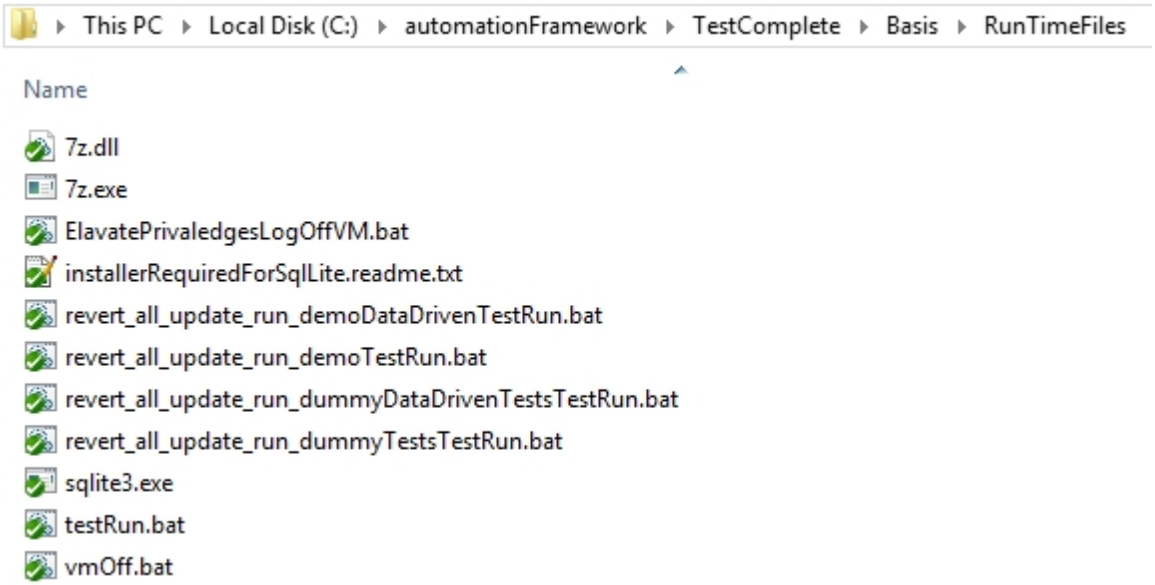
Executing Overnight Test Runs

Running TestComplete from the command line is quite simple and well documented. This is the basis for running over night test runs on physical or virtual machines.

How to Execute a Test Run

The steps for starting an overnight test run are as follows:

1. As TestComplete relies on windows handles you need to be logged in to the host machine via RDP prior to starting the test run.
2. From within the RDP session browse to the <RunTimeFiles> directory and simply execute the desired test run batch file this will have the name: `revert_all_update_run_[Test Run Name].bat`



3. Watch for TestExecute running
4. After 30 seconds your session will terminate

What Happens When a Test is Run?

When one of the test run batch files is executed the following happens:

1. The version control system is rolled back any changes on the VM are discarded and the latest update is fetched
2. The test run with the matching name is launched via TestExecute
3. A batch file that sleeps for thirty seconds and steals the RDP session is launched
4. The test run will keep running in this session until it is finished

How Do These Batch Files Get Generated?

Generating these batch files is simply a matter of running the `deleteRegenerateBatchFiles` function in the `main` script. This function will generate a batch file for every function in main that ends in the test: `TestRun.` e.g. `function demoDataDrivenTestRun().`

These files will be added to the RunTimeFiles directory and can be checked into version control from there.

Extensions

As the test suite gets bigger this process can be extended and refined. Such extensions could include:

1. Sending email notifications - there is an `emialUtils` script in the framework which can be used but getting this to work may require technical support as often environments are configured to block such traffic. Firewall and or email servers may need to be configured
2. Running a test on a windows scheduler - may need an utility like caffeine to keep the session open when tests are not running

Cross Browser Testing

Running Different Browsers

Cross browser testing is achieved from within the Zenith Framework via `runConfig` parameter `targetBrowserName` e.g:

```
function defaultRunConfigInfo(){
  fullyEnableCallStack();
  notImplementedWarning('Only calling default - defaultRunConfigInfo - update
this method as required');
  var result = {
      requiredProperties:['name'],
      mocking: false,
      mockReplace: MOCK_ADD_MISSING_REPLACE_FAILED(),
      demo: false,
      country: 'Australia',
      targetBrowserName: BROWSER_NAME_FIREFOX(),
      // an empty array in the tests property
      // means the test list will be ignored by default
      tests: []
  }
  return result;
}
```

To run the same test run with different browsers simply use a different `targetBrowserName` in the `runConfig` for the test Run or TestCaseEndPoint:

Example: TestRun

```
function demoDataDrivenIETestRun(){
  runAcmeTests(
                {
                  name: 'Data Driven Tests Demo',
                  tests: ['*Deferr*', '*HOF*'],
                  targetBrowserName: BROWSER_NAME_IE(),
                  demo: true
                }
               );
}
```

An IE test run at the Acme company

Example: TestCaseEndPoint

```
function testCaseEndPoint(){
  var testParams = {
```

```
                    testConfig: TEST_CONFIGURATION,
                    itemSelector: all,
                    mocking: false,
                    targetBrowserName: BROWSER_NAME_IE(),
                    demo: true
                   };
  runTestCaseEndPoint(testParams);
}
```

Setting an Overall Default Browser

The site wide (i.e. for everyone using the project) default browser can be set by changing the `defaultBrowser()` function in `WebUtilsPrivate`.

Example

```
function defaultBrowser(){
  return btFirefox;
}
```

A Final Note

Most scripts will run across browsers when interacting with the DOM. Scripts will often need to be customised, however, when dealing with browser specific (outside the DOM) issues like closing windows dialogue boxes or switching tabs.

Setting Up an Environment

This section is mostly intended as a brief introduction for new consultants. It may also be of interest to QA who wish to develop a broader understanding of what makes the Zenith Framework work. It briefly describes the main steps to getting started on a new site.

Test Case Selection and Modification

A reasonable number of test cases need to be selected and modified so they are suitable for automation. These test cases should be:

Broad - they should cover as many areas of the application as possibly without necessarily being too deep. The initial test cases should cover as much of the AUT as possible because this gives the highest chance of discovering any difficult to automate areas early in the process when an experienced consultant is present

Concise - UI automation tests are fundamentally different to manual tests. UI automation tests should do the minimum interaction with the AUT required to test a particular feature. Because manual testing is so resource intensive manual tests tend to be long and rambling. They tend to test as much functionality as possible within each stage of a test because the effort involved in navigating to a particular location in the AUT is so high it is best to make the most opportunity to test a particular screen once the user is there. In automated tests the costs are related to the time taken to rerun a particular scenario when a test has failed. For this reason a lot of small targeted tests that run quickly is far better small numbers of large tests with long run times

Clear - The consultant should not have to spend a large amount of time interpreting the test cases. The test cases should not assume large amounts of domain knowledge

Licenses

The correct licenses need to be procured early in an engagement. It it usually the case that enterprise floating licenses are the best value. These allow anyone in an organisation to use TestComplete but limits the number of concurrent users. Note also that can be invaluable for overnight testing and allowing a test run to proceed while all TestComplete licenses are being used. See the SmartBear Web Site for details.

Setting Up the Development and Run Environments

TestComplete must be set up to run robustly and overnight runs must be set up very early in an engagement. Ideally even before any non trivial test cases are written. The issues below, which require the most input from the client, are easily deferred, but doing so can put the success of the overall engagement at risk.

Rollover

Rollover is vital to simplifying the creation of automated test cases. The rollover should delete and replace or rollback the data source of the AUT and reset any other data, such as configuration data that may have been changed by a test. There may also be alternative starting data sources for different tests. It is best that the run environment be maintained on the executing client not shared between clients. If, for example, there was a database dedicated to test automation then only one person at a time could develop and debug test scripts and a test run could not be executed at the same time tests were being maintained. This is because if a user rolled over the database then this could break the test run or interfere with another user.

What happens when a rollover is executed will depend on the architecture of the AUT. It usually requires replacing or rolling back a database to a snapshot and will require the assistance of developers to implement. Note that TestComplete can call static functions in dot net assemblies, java classes and DLLs so the developer could use any one of these types of executables to implement rollover. Also batch files can be called simply from TestComplete using framework utility functions such as executeFile.

Once the rollover has been implemented it should be called from the rollOver function in restart scripts.

Dedicated Machine(s) or VM(s)

It is important to have a fully configured environment for overnight test runs, these need to be on a dedicated machine(s) or VMs. Note that by default you cannot run TestComplete on virtual machine unless there is a remote connection open from a client machine and this window cannot be minimised. There are, however, work arounds for both these issues so you are able to run tests with the remote desktop window minimised or you can completely disconnect from the session. See Executing Overnight Test Runs for details.

Do not fall into the trap of building a test suite without getting the overnight environment set up running every night. Resolving the issues and setting down procedures related to running and maintaining the tests is as important as test suite itself. The earlier this is done the easier it is.

Version Control

The seed project and documentation needs to be added to version control. Ensure that the correct files are ignored (see .hgignore and add the same patterns to the company's version control system).

Batch Start

Related to setting up a dedicated overnight run environment is creating a batch file to kick off the test run itself. See Executing Overnight Test Runs for further details.

Setting Up TestComplete

Configure TestComplete (on all workstations) as follows:

1. Open the Seed project suite provided
2. Import the provided settings file: Settings.acnfg: <Tools><Settings><Import Settings...>. After

you have done this make sure the <Units encodeing> when you <Right Click><Edit><Project Properties> is ANSI.

3. Import code templates as follows:
    1. In the project workspace <Double Click> any script
    2. <Right Click> anywhere in the script editor
    3. <Panel Options>
    4. select <Panels><Coded Editor><Code Templates> then
    5. click the <Load from File> button
    6. Select the codeTemplates.ct file provided

Setting Up the Framework

Once the environment is set up and you have opened the Seed project there are a number of changes that need to be made to get the Zenith Framework up and running on site.

Copyright Notice

There is a Copyright notice at the bottom of each file these need to be updated with the company name: replace THE-COMPANY with the actual company name throughout the whole suite.

Custom Functions

The way the framework runs is customised by providing custom implementations to the sub-functions that are called when a test run is executed. There is also another function that needs to be customised before the framework can be used to run meaningful tests.

If you run the demo test cases in the Seed project, (by executing dummyTests() in Main) the project should run successfully. There will be some deliberate errors that have been placed in the dummy tests for demonstration purposes but, more importantly, there will be a number of warnings in the log file. These warnings will all end in the text: NOT IMPLEMENTED and indicate incomplete implementation of site specific functions. When implementation is complete then there will be no NOT IMPLEMENTED warnings anywhere in a test run.

The first hint as to what needs customisation can be gained by looking in MainPrivate in the function called: run_THE_COMPANY_NAME_Tests this would be renamed based on the company name:

```
function runAcmeCorporationTests(configFileNoDirOrConfigObj){
  runTests(
    configFileNoDirOrConfigObj,
    defaultRunConfigInfo(),
    defaultTestConfigInfo(),
    allFilters,
    simpleLogProcessingMethod
  );
}
```

The first parameter is a config file or object the others are methods that all need to be modified on a site by site basis: `defaultRunConfigInfo`, `defaultTestConfigInfo`, `simpleLogProcessingMethod`.

- `defaultRunConfigInfo` - returns an object with all required and optional fields in a run configuration see Run Step Details
- `defaultTestConfigInfo` - returns an object with all required and optional fields in a test configuration see Run Step Details
- `simpleLogProcessingMethod` - By the end of a test run a simplified log file has been generated. Currently this is simply parsed and the results added to the TestComplete test log. More functionality can be added to this method, such as emailing the results to the QA manager.

Presentation and Self Training Scheduled

Once the above functions have been implemented the consultant is ready to start implementing test cases. This stage of the framework implementation is known as the Initial Automation Period. This is normally a period of three or four weeks which the consultant dedicates to implementing a broad range of test cases prior to the commencement of mentoring and training. During this time the consultant should to ensure all non-trivial site specific issues are resolved.

The consultant should seek an undertaking by those who are to undergo training to read this documentation and start experimenting with the framework prior to the end of the Initial Automation Period. A presentation (included with this framework) should be held for all stakeholders at the end of the Initial Automation Period and should be immediately followed by the start of mentoring and training.