



Quick answers to common problems

# TestComplete Cookbook

Over 110 practical recipes teaching you to master TestComplete – one of the most popular tools for testing automation

**Gennadiy Alpaev**

[www.allitebooks.com](http://www.allitebooks.com)

**[PACKT]**  
PUBLISHING

# TestComplete Cookbook

Over 110 practical recipes teaching you to master TestComplete – one of the most popular tools for testing automation

**Gennadiy Alpaev**



BIRMINGHAM - MUMBAI

# TestComplete Cookbook

Copyright © 2013 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: December 2013

Production Reference: 1091213

Published by Packt Publishing Ltd.  
Livery Place  
35 Livery Street  
Birmingham B3 2PB, UK.

ISBN 978-1-84969-358-5

[www.packtpub.com](http://www.packtpub.com)

Cover Image by Aniket Sawant ([aniket\\_sawant\\_photography@hotmail.com](mailto:aniket_sawant_photography@hotmail.com))

# Credits

**Author**

Gennadiy Alpaev

**Project Coordinator**

Priyanka Goel

**Reviewers**

Mykola Kolisnyk

Giri Prasad Palanivel

Drashti Pandya

Rakesh Kumar Singh

**Proofreader**

Jenny Blake

**Indexer**

Mehreen Deshmukh

**Acquisition Editors**

Amarabha Banerjee

Usha Iyer

Nikhil Karkal

**Graphics**

Sheetal Aute

**Production Coordinator**

Conidon Miranda

**Lead Technical Editor**

Priya Singh

**Cover Work**

Conidon Miranda

**Technical Editors**

Tanvi Bhatt

Zainab Fatakdawala

Pankaj Kadam

Hardik B. Soni



# About the Author

**Gennadiy Alpaev** has been working as a test automation engineer since he graduated in 2003. During these 10 years he has worked with many automation tools including TestComplete, SilkTest, Selenium, and Squish and participated in more than 10 projects, both as an employee and on contract basis.

He gained his first experience of writing tutorials when he created a tutorial on SilkTest together with his colleague Mykola Kolisnyk. His second big project on test automation was a complete tutorial on TestComplete. These two tutorials are available online in Russian.

Starting in 2011, he is running online and on-site courses on TestComplete and test automation for independent students and companies. He is also actively participating in different forums trying to help others solve problems as fast as possible.

The main principles that he follows in his training are simplicity and brevity, especially when explaining complex things.

---

First of all I'd like to thank Andrew Grinchak who helped me with the English version of the book. Without him the work would have been much more complicated.

Many thanks to all forum participants for their questions and answers. There are so many of them that it is impossible to mention them all, but without them I would have never got all the experience I have now. Also during these 10 years, I had great pleasure working with different people who influenced my vision and qualification. Thank you all, every communication was helpful and interesting, and introduced its part to this book.

And finally thanks to my family for their patience and support during the process of writing this book.

---

# About the Reviewers

**Mykola Kolisnyk** has been in test automation since 2004, being involved in various activities including creating test automation solutions from scratch, leading test automation team, and performing consultancy regarding test automation processes. During his working career he has worked with different test automation tools such as Mercury WinRunner, MicroFocus SilkTest, SmartBear TestComplete, Selenium-RC, WebDriver, SoapUI, BDD frameworks, and many other different engines and solutions. He has also had experience with different domain areas such as healthcare, mobile, telecom, social networking, business process modeling, performance and talent management, multimedia, e-commerce, and investment banking.

He has worked as the permanent employee at ISD, GlobalLogic, and Luxoft, is experienced in freelance activities, and was invited as an independent consultant to introduce test automation approaches and practices to external companies.

He's one of the authors (together with Gennadiy Alpaev) of online SilkTest Manual (<http://silktutorial.ru/>) and has participated in the creation of TestComplete tutorial (<http://tctutorial.ru/>) which are some of the biggest related documentation available in RU-net.

**Giri Prasad Palanivel**, a Senior Test Automation Engineer, has expertise in test automation, agile, and performance testing. He is an enthusiast and explorer, and is passionate about working with the latest technologies and tools. His areas of interest are designing automation frameworks and managing overall QA activities. He believes in collaborative learning, contributes to testing community, and mentors juniors.

---

I wish to thank all those who, in various ways, have helped me in my career and special thanks to my family for their support and care.

---

**Drashti Pandya** is a Bachelor of Engineering from Mumbai University. She has 5 years of experience in IT industry with expertise on Software Testing.

---

I would like to thank my family for their support.

---

**Rakesh Kumar Singh** is a QA Manager at Airpush, Inc. with more than 7 years of experience in the field of software quality assurance. He has worked in USA and India with majority of work experience in USA.

He has done his Master of Science in Computer Engineering from California State University, Chico in USA and Bachelor of Engineering in Electronics and Communications from VTU, Karnataka in India.

He has worked in several fields of quality assurance including manual, automation, performance, and load testing.

He has worked in companies such as Airpush Inc, Jutera Inc, Dynamic Logic—a WPP company, BlackRock Solutions, and GSI Commerce—an Ebay company.

---

I would like to thank my family.

---

# www.PacktPub.com

## Support files, eBooks, discount offers and more

You might want to visit [www.PacktPub.com](http://www.PacktPub.com) for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at [www.PacktPub.com](http://www.PacktPub.com) and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at [service@packtpub.com](mailto:service@packtpub.com) for more details.

At [www.PacktPub.com](http://www.PacktPub.com), you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<http://PacktLib.PacktPub.com>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

## Why Subscribe?

- ▶ Fully searchable across every book published by Packt
- ▶ Copy and paste, print and bookmark content
- ▶ On demand and accessible via web browser

## Free Access for Packt account holders

If you have an account with Packt at [www.PacktPub.com](http://www.PacktPub.com), you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.



# Table of Contents

<b>Preface</b>	<b>1</b>
<b>Chapter 1: Getting Started</b>	<b>7</b>
Introduction	7
Installing TestComplete	8
Creating your first project	11
Choosing a scripting language for the project	13
Recording your first test	15
Modifying the recorded test	18
Saving and restoring user settings	20
Creating code templates	22
Delaying script execution	24
Adding and removing project items	25
Understanding how TestComplete interacts with tested applications	27
Understanding Object Browser	29
Using Object Spy	32
<b>Chapter 2: Working with Tested Applications</b>	<b>35</b>
Introduction	35
Adding a tested application project item	36
Running a tested application from the script	38
Terminating a tested application	40
Killing several instances of a tested application	41
Closing a tested application	43
Running a tested application in the Debug mode	45
Running a tested application under a different user account	47
Changing tested application settings dynamically from the script	49
Running external programs and DOS commands	50
Testing installers – running an MSI file	52



<b>Chapter 3: Scripting</b>	<b>55</b>
Introduction	55
Entering text into text fields	56
Using wildcards to process objects with variable names	58
Structuring code using loops	61
Creating property checkpoints	64
Creating object checkpoints	68
Using global variables	73
Testing multilingual applications	75
Working with nonstandard controls	79
Organizing script code in the project	83
Handling exceptions	86
Handling exceptions from a different unit	89
Creating framework using the OOP approach	92
<b>Chapter 4: Running Tests</b>	<b>97</b>
Introduction	97
Running a single function	98
Verifying test accuracy	100
Creating a test plan for regular runs	101
Running tests from the command line	103
Passing additional parameters to test from the command line	106
Organizing test plan runs	109
Scheduling automatic runs at nighttime	110
Running tests via Remote Desktop	112
Changing playback options	113
Increasing run speed	115
Disabling a screensaver when running scripts	116
Sending messages to Indicator	118
Showing a message window during a script run	119
<b>Chapter 5: Accessing Windows, Controls, and Properties</b>	<b>121</b>
Introduction	122
Choosing Object Tree Model	122
Understanding the window's life cycle	124
Ignoring overlapping windows	125
Dragging one object into another	127
Calling methods asynchronously	131
Verifying if an object has a specific property	132
Finding objects by properties' values	134
Waiting for an object to appear	136

---

Waiting for a property value	139
Mapping custom control classes to standard ones	140
Using text recognition to access text from nonstandard controls	143
Using Optical Character Recognition (OCR)	145
Dealing with self-drawn controls not supported by TestComplete	149
<b>Chapter 6: Logging Capabilities</b>	<b>153</b>
<hr/>	
Introduction	153
Posting messages to the log	154
Posting screenshots to the log	156
Creating folders in the log	158
Changing log messages' appearance	159
Assessing the number of errors in the log	161
Changing pictures' format	163
Comparing screenshots with dynamic content	165
Decreasing log size	167
Generating log in our own format	168
Exporting log to MHT format	170
Sending logs via e-mail	171
<b>Chapter 7: Debugging Scripts</b>	<b>175</b>
<hr/>	
Introduction	175
Enabling and disabling debugging	176
Using breakpoints to pause script execution	177
Viewing variables' values	179
Debugging tests step by step	181
Evaluating expressions	182
<b>Chapter 8: Keyword Tests</b>	<b>185</b>
<hr/>	
Introduction	185
Recording and understanding Keyword Tests	186
Adding new actions to existing Keyword Tests	188
Enhancing Keyword Tests using loops	191
Creating object checkpoints	193
Calling script functions from Keyword Tests	195
Converting Keyword Tests to scripts	198
Creating our own Keyword driver	199
<b>Chapter 9: Data-driven Testing</b>	<b>203</b>
<hr/>	
Introduction	203
Generating random data for tests	204
Accessing a specific cell in a table	206
Reading all data from a table	208

Using DDT tables for storing expected values	209
Changing CSV delimiter and other parameters	212
Driving data without using loops	214
Accessing Excel spreadsheets without having MS Office installed	215
Auto-detecting Excel driver	216
<b>Chapter 10: Testing Web Applications</b>	<b>217</b>
Introduction	217
Choosing Web Tree Model	218
Using updates for the latest browser versions	221
Performing cross-browser testing	222
Verifying if a text exists on a page	224
Waiting for an element to appear on a page	225
Saving screenshots of an entire page	228
Running scripts on a page	229
<b>Chapter 11: Distributed Testing</b>	<b>231</b>
Introduction	231
Setting up Network Suite and understanding distributed testing	232
Copying Project Suite to a Slave workstation	234
Using a Master workstation to run tests	235
Using different configuration files for each workstation	236
Sharing data between workstations	237
Synchronizing test runs on several workstations	239
<b>Chapter 12: Events Handling</b>	<b>241</b>
Introduction	241
Creating event handlers	242
Disabling the postage of certain error messages	244
Clicking on disabled controls without an error message	245
Handling unexpected windows that affect TestComplete	248
Handling unexpected windows that don't affect TestComplete	249
Saving the log to a disk after each test	251
Sending a notification e-mail on timeouts	253
Creating preconditions and postconditions for tests	254
<b>Index</b>	<b>257</b>

# Preface

Testing automation is a tricky and complex area of computer science, as it requires not only experience in both testing and programming, but also knowing some specifics of using Graphical User Interface (GUI).

During the last several years a lot of software has been created to help to automate testing by emulating users' actions. Some of these programs are strictly specialized, while others allow users to automate a wide range of software.

TestComplete is one of the tools which supports testing of software developed on different platforms and application types (.NET, Win32, Java, Delphi, Web, and so on), at the same time using similar techniques for all of them, thus simplifying process of automation by software testers.

This book will teach you how to effectively use TestComplete by many simple and well thought-out examples, at the same time showing how to solve the most frequently asked questions. By executing the steps from each recipe and then reading the explanation text of what has been done, you will master TestComplete quickly and easily.

We hope this book will be a great support to you in studying TestComplete and testing automation principles.

## **What this book covers**

*Chapter 1, Getting Started*, provides basic information about TestComplete and prepares you for further topics. This chapter will be helpful if you are new to TestComplete.

*Chapter 2, Working with Tested Applications*, explains how to work with tested applications in TestComplete using different approaches.

*Chapter 3, Scripting*, provides programming solutions for frequently asked questions and shows examples of different testing methodologies and frameworks.

*Chapter 4, Running Tests*, explains how to run your TestComplete tests including running from command line and scheduling automatic test runs.

*Chapter 5, Accessing Windows, Controls, and Properties*, explains how TestComplete interacts with tested applications, their controls, and data within windows.

*Chapter 6, Logging Capabilities*, covers several topics related to TestComplete log including working with screenshots, exporting logs, and sending results via e-mail.

*Chapter 7, Debugging Scripts*, describes how to use TestComplete debug capabilities when maintaining tests.

*Chapter 8, Keyword Tests*, introduces a simple way to create automated tests which doesn't require programming skills.

*Chapter 9, Data-driven Testing*, explains how to separate scripts code from test data and effectively work with it in tests.

*Chapter 10, Testing Web Applications*, covers Web-specific topics which were not covered in other chapters.

*Chapter 11, Distributed Testing*, shows how to run tests on several workstations and share data between them.

*Chapter 12, Events Handling*, introduces TestComplete events—a powerful tool to customize and improve your testing framework.

## **What you need for this book**

To run the examples in the book the following software will be required:

- ▶ Microsoft Windows 7 x86 (32 bits) Home, Professional, Ultimate, or Enterprise Edition (most examples will also work in 64-bits version of the OS)
- ▶ SmartBear TestComplete 9.0 or higher (most examples will also work in TestComplete 7.x and 8.x)
- ▶ Microsoft Calculator Plus 1.0
- ▶ Mozilla Firefox (any version supported by selected TestComplete version)

## **Who this book is for**

If you wish to use TestComplete for testing automation, this book will help you learn the very basics of the tool, as well as improve your knowledge if you already have some experience in working with TestComplete.

The recipes provided in this book can be studied one by one, thus improving your knowledge step-by-step. The book can also be read randomly, depending on the type of problem that you are trying to resolve.

It is implied that you are already aware of the programming basics (knowing what is a variable, loop, condition, function). This will tremendously facilitate further understanding of the approaches and solutions that are being considered. Nonetheless, we have tried to select examples that would be easy to understand, even for a novice in programming. These examples are easily scalable for your specific needs, as well as portable and scalable.

## Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles, are shown as follows: "The `Runner.CallObjectMethodAsync` method accepts two parameters: the `object` and the `callee` method."

A block of code is set as follows:

```
function testPirtureFormat ()
{
    Options.Images.ImageFormat = "BMP";
    Log.Picture(Sys.Desktop.Picture(), "BMP screenshot");
    Options.Images.ImageFormat = "PNG";
    Log.Picture(Sys.Desktop.Picture(), "PNG screenshot");
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

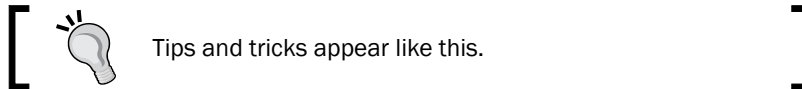
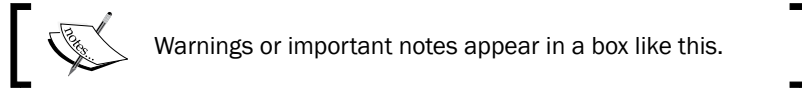
```
TestedApps.notepad.Close();
if(pNotepad.WaitWindow("#32770", "Notepad", -1, 1000).Exists)
{
    pNotepad.Window("#32770", "Notepad").Keys("~n");
}
```

Any command-line input or output is written as follows:

```
C:\Program Files\SmartBear\TestComplete 9\Bin\TestComplete.
exe" "z:\TestCompleteCookBook\TestCompleteCookBook.pjs" /run /
project:Chapter4 /Unit:Unit1 /routine:testDemoTestItems2
```



**New terms** and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Click on the **Configure** button in the **Images** group".



## Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to [feedback@packtpub.com](mailto:feedback@packtpub.com), and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on [www.packtpub.com/authors](http://www.packtpub.com/authors).

## Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

## Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

## Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from <http://www.packtpub.com/support>.

## Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at [copyright@packtpub.com](mailto:copyright@packtpub.com) with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

## Questions

You can contact us at [questions@packtpub.com](mailto:questions@packtpub.com) if you are having a problem with any aspect of the book, and we will do our best to address it.



# 1

## Getting Started

In this chapter we will cover the following recipes:

- ▶ Installing TestComplete
- ▶ Creating your first project
- ▶ Choosing a scripting language for the project
- ▶ Recording your first test
- ▶ Modifying the recorded test
- ▶ Saving and restoring user settings
- ▶ Creating code templates
- ▶ Delaying script execution
- ▶ Adding and removing project items
- ▶ Understanding how TestComplete interacts with tested applications
- ▶ Understanding Object Browser
- ▶ Using Object Spy

### Introduction

TestComplete is one of the most popular commercial tools for automation testing. It allows us to automate testing of wide variety different applications (such as Win32, .NET, Java, Web, Delphi, and many others) using the same testing methodologies and similar programming approaches.

TestComplete supports testing on all Windows platforms being supported by Microsoft (both 32- and 64-bit configurations) and allows using several scripting languages for creating scripts as well as nonscripting tests for users who do not have programming experience (keyword-driven testing).

Recommended system requirements for TestComplete are:

- ▶ Processor: Intel Pentium 4 (3 GHz) or Intel Core 2 Duo (2 GHz or higher)
- ▶ RAM: 2 GB
- ▶ HDD: 700 MB free space
- ▶ Monitor: 1280 x 1024 or higher resolution

You can find the full list of requirements on the SmartBear website (<http://smartbear.com/products/qa-tools/automated-testing-tools/testcomplete-specifications/testcomplete-system-requirements>).

In this book we use the following configuration:

- ▶ Microsoft Windows 7 x86 (32-bits) operating system installed on a virtual PC
- ▶ 1 GB of RAM
- ▶ TestComplete Version 9.30 (most examples will also work in TestComplete 7.x and 8x)

Most examples in this book use standard Windows applications (Notepad, Internet Explorer, and Paint). Additionally, you will need to install free Microsoft Calculator Plus application, which can be downloaded from the Microsoft's website (<http://www.microsoft.com/en-us/download/details.aspx?id=21622>). Standard calculator application of the Windows Vista/7/8 will not work as its controls are recognized differently and require additional adjustment.

In this chapter, we are going to cover the basic actions one can perform with TestComplete in order to get acquainted with TestComplete IDE and its tools.

## Installing TestComplete

Before getting down to TestComplete, make sure it is installed to begin with.

Installing TestComplete is quite simple; it is no different from installing most Windows applications.

### Getting ready

To install TestComplete, we will need to download installation file from the SmartBear website:

- ▶ If you are up for using a trial version of TestComplete, please follow the link <http://smartbear.com/products/qa-tools/automated-testing-tools/free-testcomplete-trial>, enter all the required data (name, e-mail, company, and so on), and click on the **Sign Up Now** button. Thus, you will receive a letter with a download link and activation code for a 30-days trial version of TestComplete.

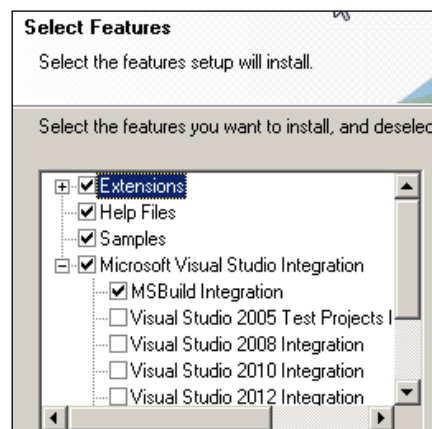
- ▶ If you are registered as a SmartBear client and have an account on their website, please follow the link <https://my.smartbear.com/login.asp>, enter your account data, and click on the **Login** button. After this, you will be redirected to the page with available programs to download and install. Clicking on the link with the version near the TestComplete, you will find yourself on the page for the download with the link and the license key.

After this you will have an installation file named `TestCompleteXYY.exe` (where `XYY` stands for the current TestComplete version).

## How to do it...

To accomplish successful TestComplete installation, follow these steps:

1. Launch the downloaded `.exe` file and wait for the **InstallShield Wizard** window to appear.
2. Click on **Next**.
3. In the **License Agreement** window, check the **I agree...** option and then click on **Next**.
4. In the **Customer Information** window enter a user name and that of a company (it may be arbitrary, not necessarily corresponding with the information from the TestComplete license). Select the type of installation (for any users or only for the current one), and then click on **Next**.
5. In the **Choose Destination Location** window, select the destination folder to install TestComplete and then click on **Next**.
6. In the **Select Program Folder** window, signify the folder in the main menu to target the shortcut of the program and click on **Next**.
7. In the **Select Features** window, select necessary components to install and click on **Next**. If you are not sure which components to install, leave all the options by default.





8. Click on **Next** in the **Start Installation** window and wait for the installation to complete.
9. Click on **Finish**.
10. Now, all we have got left to do is activate the TestComplete. In order to do so launch TestComplete application.
11. In the new popped-up window, click on the **Activate license** button.
12. Check the **Automatic activation** option and then click on **Next**.
13. Enter your data (name, company, e-mail, and key) that you signified for the registration and activation code that you have received by the e-mail. Click on **Next**.
14. If necessary, input the parameters of your proxy server and click on the **Activate** button.
15. In a matter of several minutes, necessary to connect to the server and validate registration data, you will receive the notification: **Your license has been activated successfully. Thank you..** Click on the **Finish** button. TestComplete will fire up to the starting pane. Now you are able to begin your working expertise.

### How it works...

During the installation process TestComplete verifies programs installed on your computer. The **Select Features** window displays only components which are installable at the moment. All other features (which are not accessible at the moment) will be turned off.

So far, we have considered the simplest method of activation: automatic activation of the trial TestComplete license on the SmartBear server. Automatic activation of the node-locked license is not complex either.

TestComplete uses two license types:

- ▶ **Node-locked license:** This type of license is attached to one computer and cannot be used on virtual machines
- ▶ **Floating license:** This type of license allows running several copies on multiple workstation in the local network (the number of copies is determined by the license key) and can be used on virtual machines

### There's more...

If you are occupied as a system administrator in your company and need to install TestComplete on several computers, you can cut back on the installation time by using silent installation mode from command prompt.

In order to do that, you should extract the content of the downloaded archive `TestCompleteXYX.exe` and launch the installation file `Setup.exe` with the `/r` parameter:

```
<path_to_extracted_files>\Setup.exe /r
```

The downloaded archive file can be opened with the help of any archiver that supports RAR SFX formats (for example, WinRAR or 7-Zip).

Having done so, you should carry out the previously listed steps. As a result, there will appear the `Setup.iss` file in the `Windows` folder. Move this file to the folder with the files for TestComplete installation and copy the content of the folder to the targeted computers for TestComplete installation. Launch the installation file with the `/s` parameter:

```
<path_to_extracted_files>\Setup.exe /s
```

As a result of this TestComplete will be installed with the same settings as the first installation.

## See also

- ▶ In the event of a floating license activation as well as license activation on a computer that is not connected to the Internet, it is recommended that you read up on SmartBear article via the following link:

<http://support.smartbear.com/viewarticle/33840/>

## Creating your first project

Similar to many other IDEs, TestComplete binds all the elements together in a single project; and the projects, in their due turn, are joined into project suites.

First and foremost, it is necessary to create a project suite and then create one or more projects in it to be able to add all the necessary elements for the project.

## How to do it...

In order to create a project we need to perform the following steps:

1. Select the following menu item **File | New | New Project Suite**.
2. In the **Create Project Suite** window signify the name of the project suite and the path to it.
3. Click on **OK**. In the result, there will appear the created project suite on the **Project Explorer** panel in the left part of the TestComplete window.

- Right-click on the created project suite and select menu item **Add | New Item**, as shown in the following screenshot:



- In the opened **Create New Project** window enter the name of the project and the path to it, as you have just done in the previous instance.
- In the **Language** drop-down menu select the necessary programming language of your choice and click on the **Create** button (if you do not know which language to choose, please read the *Choosing scripting language for the project* recipe).
- In the result, a project will be created into which we will be able to write the testing scripts.



You can also start creating a new project without creating a project suite first. In this case project suite will be created automatically.



## How it works...

Each project suite and project has corresponding files in the XML format with the extensions of `.pjs` and `.mds` respectively. In these files, all the necessary information is stored: elements that are constituent parts of the project at hand or a number of projects, paths to them, their parameters, and so on.

Usually, the folders with the projects are stored in the same folder with the project suite. This is quite handy since the same TestComplete structure is stored on your **hard disk drive (HDD)** as well.

If, in any event or reason, you need to store the project separately from the project suite where it is located, it is sufficient at the point of creation to signify a different path. Please note, however, that in case of moving projects to another computer in the future, you will come up against another problem: migrating all the projects together with the project suite, which are located in different folders or even on separate discs. In this case, when opening the project suite, TestComplete will prompt you with an error message **Project not found**.

## There's more...

Try to give the projects and project suites some sensible names; do not use the default names (`Project1`, `Project2`, and so on). Otherwise, you will forget which project stands for which data.

The names of the projects and project suites are also used at the point of launching tests from the command prompt.

## Choosing a scripting language for the project

Choosing a scripting language for the project is the first important choice to make before creating a project. Choosing a language should be a careful and circumspect process, since it shall not be possible to change the choice in the future. If you would like to change the selected project's scripting language, you would have to redo the project from scratch!

## How to do it...

In order to select a scripting language for a new project we need to perform the following steps:

1. Start creating a new project (by selecting **File | New | New Project**).
2. In the **Language** combobox, you will see a list of five languages available.
3. Select one of them depending on your needs.

## How it works...

TestComplete provides a possibility of choice from the following three programming languages: **JScript**, **VBScript**, and **DelphiScript**. Apart from these three languages, the following two are also available: **C++Script** and **C#Script**. The latter two languages are in fact the same as JScript, with somewhat modified syntax. That's why everything that goes for JScript is just as applicable for these two scripting languages also.



The C++Script and C#Script scripting languages have nothing in common with C++ and C#! It's the same JScript with a slightly changed syntax. By using C#Script you will not have the possibilities extended in the C#! The same goes for C++Script.

The next important thing: if you are planning to create tests only in TestComplete and then launch them with the help of TestComplete (or with the help of **TestExecute** – a command-prompt utility), you can select absolutely any language, regardless of the application that you are about to test.

For example, you may use VBScript language to test applications coded in C#, or select DelphiScript to test web-applications. In any case, you will enjoy complete access to all the TestComplete possibilities. For example, to access standard .NET classes in TestComplete, there is a special `dotNET` object up for grabs. This object can be used in any programming language.

If you are already familiar with one of the languages suggested by TestComplete, selecting just that will be better for you. If none of the languages are familiar to you, the following tips may come in mighty handy:

- ▶ **VBScript:** This language is very simple to learn, and therefore is recommended for beginners who are not proficient in programming.
- ▶ **JScript:** This language (JavaScript engine from Microsoft) is a more powerful and flexible language in comparison to the VBScript; it also has more compact syntax and its code constructions are shorter and faster to type. JScript is recommended for those who have some programming background.
- ▶ **DelphiScript:** This language is a procedural scripting language used only in TestComplete. Its syntax resembles a skimpy version of Delphi. It is recommended only for creation of connected and self-tested applications (see later in this chapter).

### There's more...

Now, we shall consider why we need such languages as DelphiScript, C++Script, and C#Script.

TestComplete allows scripting in more advanced languages (C#, C++, Delphi, and Visual Basic). Meanwhile, you are using all the functionalities of a given language, writing up tests in any appropriate IDE, using the functionalities extended by TestComplete to gain access to the tested application. If you plan to record scripts with the help of TestComplete first and then convert them to tests in more advanced languages, you will need to apply these three languages. You can select a scripting language to comply with what it will be converted to (for example, C++Script to convert to C++code, DelphiScript for converting to Delphi). Having resolved that, the process of converting becomes hands-down easy. You will only need to make several similar changes in the code. In other cases, usage of the languages DelphiScript, C++Script, and C#Script is usually not considered expedient.

### See also

- ▶ Of course, each language has its flaws. A complete listing of limitations can be found at <http://support.smartbear.com/viewarticle/32212/>.
- ▶ Differences in the scripting tests and tests written in advanced-level languages is explained at <http://support.smartbear.com/viewarticle/27178/>.

## Recording your first test

Recording is the simplest way to create your first auto test. No programming skills are required as it is extremely simplistic to go about and execute. In this recipe, we will make a recording of the first executable test to be launched to make sure it is workable.

### Getting ready

Before recording the first test, we need to perform some prerequisite steps:

1. Create a new project in TestComplete, as described in the *Creating your first project* recipe.
2. Download the Calculator Plus application from the Microsoft website, if you have not yet done so, and install it (<http://www.microsoft.com/en-us/download/details.aspx?id=21622>). This application will be needed for many examples in the book.
3. Launch the Calculator Plus and opt for the **View | Classic View** menu, so that the Calculator Plus could look like a no-frill Windows application, without bells and whistles.
4. Select the **View | Standard** menu item. Having done so, Calculator is switched to the Standard working mode.

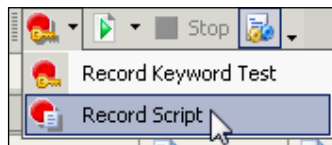


If you are working in the Windows XP or Windows Vista operation systems, you can get by with a usual calculator that comes along as an embedded system component. For Windows 7 and later, the embedded calculator will not work to handle the examples at hand, because it is much harder to obtain the calculus results from its text output field.

### How to do it...

In order to record a test we need to perform the following steps:

1. Select the **Test | Record | Record Script** menu item or go for the appropriate option from the drop-down menu on the toolbox, as shown in the following screenshot:

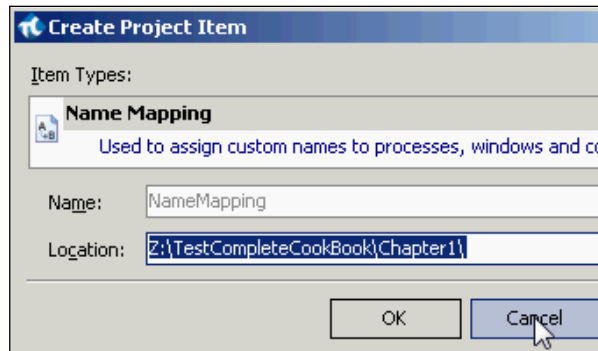


At this point, we would have a floating window widget **Recording** with the buttons **Rec.**, **Stop**, and **Pause**.

- In the Calculator Plus window, click on the buttons **2**, **+**, **2**, **\***, **5**, and **=**. In the end result, the result of the calculation will appear in the text output to the following effect:  $2+2*5 = 20$ . This is a standard calculation mode for the calculator where operation priorities are not accounted for.
- Click the **Stop** button in the **Recording** window, as shown in the following screenshot:



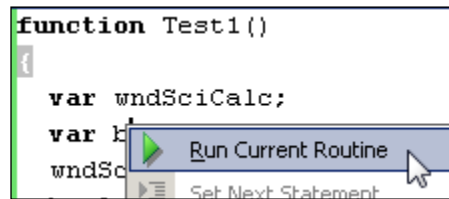
- If you have the **Create Project Item** window opened afterwards, prompting for adding the **NameMapping** element to the project, click on the **Cancel** button, as shown on the following screenshot:



- In the result, the TestComplete editor will contain the following script:

```
function Test1()
{
    var wndSciCalc;
    var btn2;
    wndSciCalc = Sys.Process("CalcPlus").Window("SciCalc",
"Calculator Plus");
    btn2 = wndSciCalc.Window("Button", "2");
    btn2.ClickButton();
    wndSciCalc.Window("Button", "+").ClickButton();
    btn2.ClickButton();
    wndSciCalc.Window("Button", "**").ClickButton();
    wndSciCalc.Window("Button", "5").ClickButton();
    wndSciCalc.Window("Button", "=").ClickButton();
}
```

6. Now, we can go ahead and double-check if the recorded script works as it's been intended. For this purpose, we right-click on any spot in the `Test1` function, and from the context menu select the **Run Current Routine** menu item:



7. As a result, the script will launch and execute all the prerecorded actions.

### How it works...

All the actions made during the script recording are transformed by TestComplete to the corresponding scripting commands, that is, mouse-clicks, text input, and selection of elements from a drop-down list; all these actions are covered by specific corresponding commands.

In our example:

- ▶ In the first two lines, there appear variables for the calculator window and the button **2**.
- ▶ In the following two lines these variables are initialized and they have specific objects assigned.
- ▶ Now, it is possible to carry out different actions with these objects.
- ▶ In the further six lines of code, we reproduce one and the same action with the help of several buttons, that is, the button-click, in particular.
- ▶ Please note that the button **2** has become peculiar. Only for this button do we have a variable (`btn2`) declared, while other buttons are handled through the window variable (`wndSciCalc`). This happens because the button **2** is being used more than once, which was duly recognized by TestComplete and further on transmuted into the recursively applied code in view.

### There's more...

Although TestComplete is generating a readable code at the point of recording, all the recorded scripts are the least readable and not easily maintainable. Sometimes, in case of changes in the tested application, the prerecorded scripts should be redone from the scratch rather than unraveled or modified to fit new conditions. Hence, the recording technicalities are not recommended to be applied to create scripts that should be workable recursively with intention to be applied for regression testing.



However, there are several cases when recording is useful, for example:

- ▶ To learn and understand how TestComplete interacts with a tested application and controls within it
- ▶ To quickly implement a simple script for a one-time task
- ▶ To record several actions for future modifications

### See also

- ▶ Recording scripts is just a first step towards creating effective scripts. To learn how to improve your tests, read the *Modifying the recorded test* recipe.
- ▶ If you want to better understand how TestComplete works with windows and controls, refer to the *Understanding how TestComplete interacts with tested applications* recipe.
- ▶ Running functions is explained in detail in *Chapter 4, Running Tests*.

## Modifying the recorded test

As we have seen in the previous recipe, a script can be recorded automatically; however, in the result of such recording, an unreadable code will be generated, which is difficult to modify.

Let's suppose that in the result of requirements for the tested application altering, we have to add 20 more button-clicks to various buttons. The simplest way is to copy the last line of code (in which the = button is clicked), however, the size of our script will increase significantly, which will worsen the readability.

In this recipe, we will modify the recorded code in such a way that we will minimize the necessitated actions to append the new Calculator Plus button-clicks.

### How to do it...

In order to modify a test we need to perform the following steps:

1. First and foremost, we will make the code conformable to a unified style so that button-clicks appear in the same way in any given case. To this end, we should get rid of the variable `btn2`; and the code that stands for the button-click should be rewritten in exactly the same manner that the button-clicks for the rest of the buttons have been coded.
2. Together with that, we will join declaration of the `wndSciCalc` variable with its initialization and assign the variable a different name. In the result, the code will appear as follows:

```
function Test1Modified1()
{
    wndSciCalc = Sys.Process("CalcPlus").Window("SciCalc",
"Calculator Plus");
    wndSciCalc.Window("Button", "2").ClickButton();
    wndSciCalc.Window("Button", "+").ClickButton();
    wndSciCalc.Window("Button", "2").ClickButton();
    wndSciCalc.Window("Button", "***").ClickButton();
    wndSciCalc.Window("Button", "5").ClickButton();
    wndSciCalc.Window("Button", "=").ClickButton();
}
```

### Downloading the example code



You can download the example code files for all Packt books you have purchased from your account at <http://www.PacktPub.com>. If you purchased this book elsewhere, you can visit <http://www.PacktPub.com/support> and register to have the files e-mailed directly to you.

3. The same repetitive actions are best to be coded as loops. In the given case, we will use the `for` loop, and the text of the major buttons will be announced as an array:

```
function Test1Modified2()
{
    var aButtons = ["2", "+", "2", "***", "5", "="];
    var wCalcPlus = Sys.Process("CalcPlus").Window("SciCalc",
"Calculator Plus");
    for(var i = 0; i < aButtons.length; i++)
    {
        wCalcPlus.Window("Button", aButtons[i]).ClickButton();
    }
}
```

4. Lastly place code for button-clicks into a separate function:

```
function clickCalcButton(caption)
{
    var wCalcPlus = Sys.Process("CalcPlus").Window("SciCalc",
"Calculator Plus");
    wCalcPlus.Window("Button", caption).ClickButton();
}

function Test1Modified3()
{
    var aButtons = ["2", "+", "2", "***", "5", "="];
    for(var i = 0; i < aButtons.length; i++)
    {
        clickCalcButton(aButtons[i]);
    }
}
```

## How it works...

We have completed the modification of the recorded code within the three steps:

- ▶ On the first step we simply arranged the code so that it followed the same style. These changes are cosmetic; we need them just to simplify the following modifications.
- ▶ On the second step we made a serious change: made an addition of the loop to iterate through the repetitive actions of the button-click. As a result, first of all, we reduced by half the size of the function; secondly, we facilitated further work. Now, if we need to add new button-clicks, it would be enough to add the heading of the button to the array, and the loop will automate any further actions with the button.
- ▶ Finally, on the third step, we have organized the calculator into a separate function. Thus, we have concealed the details of the realization of these actions by leaving only high-level actions in the testing function `Test1Modified3`. This three-step accomplishment is called functional decomposition.

## See also

- ▶ The *Organizing script code in the project* and *Creating framework using the OOP approach* recipes in *Chapter 3, Scripting*, will help you to arrange your modifications in script code

## Saving and restoring user settings

In TestComplete there are three types of customizable settings:

- ▶ Settings of the TestComplete itself (various parameters that influence regimes of work with TestComplete are available in the **Tools | Options** menu)
- ▶ Interface settings
- ▶ Project settings (in the **Tools | Current Project Properties** menu)

The first two types of the settings are easy to save and restore at any time.

## How to do it...

The following steps should be performed to save and restore settings:

1. In order to save the settings from the **Tools | Options** menu, opt for the **Tools | Settings | Export Settings** menu item, and in the opened window signify which settings you would like to save and the name of the file to save them into; then, click on the **Export** button.

2. To upload the settings saved earlier, opt for the **Tools | Settings | Import Settings** menu item, and in the opened window opt for the file of the settings and click on the **Import** button. All the settings will be uploaded in the same format they had been initially saved in.
3. To save the settings of the external view (panels and toolbars), opt for the **View | Desktop | Save Desktop As** menu item and input the name of the file which you would like to save the settings of the external view.
4. In order to upload the previously saved settings, opt for the **View | Desktop | Load Desktop** menu item, and select the earlier saved file, and then click on the **Open** button. All the settings of the interface will return to the earlier saved variation.

### How it works...

Depending on the type of settings, they will be saved into the file with different extensions (`.desktop` for the settings of the interface and `.acnfg` for the settings of the program). If you're working on different computers and wish to work everywhere with the same habitual selection of settings, you can implement this, but also make it so it only takes a couple of seconds to restore previous settings on other computers. The same can be done after TestComplete has been re-installed or in case these files are stored in the source control along with the project suite; updating local copy of the project affects TestComplete's appearance.

### There's more...

Sometimes, it comes to pass that by experimenting with the re-positioning of various panes or control panels, you cannot restore their initial positions. In order to resolve this problem, there are respective options **View | Desktop | Restore Default Docking** (restoring initial view of the panes and panels) and **View | Toolbars | Restore Default Toolbar** (restoring toolbars).

These two options can sometimes become a point-of-care rescue if you have no saved settings.

Although we have no method to save and upload settings of a separate project, we can clone the existing project (it will be copied completely, including its settings), and then remove the unnecessary elements from the new project.

To clone a project, right-click on its name and opt for the **Clone Project** menu. Then, enter the name and path for the new project and click on the **OK** button.

Apart from this, we can assign the initial settings of the project (that is, the settings by default). This can be done via the **Tools | Default Project Properties** menu option.

## Creating code templates

When coding tests, we are using different programming constructs (for example, `if...else`, `try...catch`, and so on). Some of them are used rarely, while others, conversely, are used very often.

TestComplete allows us to accelerate input of some of the programming constructs with the help of so called **code templates**.

Several of the templates are already predefined in TestComplete; however, they do not always suffice for the job. That is why we will learn how to apply the preset templates and to create some of our own.

### How to do it...

Suppose we would like to create a template of our own for the `if...else` block of JScript code:

1. Select the **Tools | Options** menu item, and in the opened dialog window, select the **Panels | Code Editor | Code Templates** option.
2. In the opened right-hand window panel select the target language that we want to add a new template for.
3. Click on the **Add** button.
4. Enter the name and description of the new template in the list of templates (to change the inputted values into the table, it is enough to make a single mouse-click on the cell).
5. In the field under the list, enter the code of the script that should correspond with the template in view.



Please take note of the symbol | (the pipeline): it defines the spot for the cursor to appear after the template has been inserted into the editor.

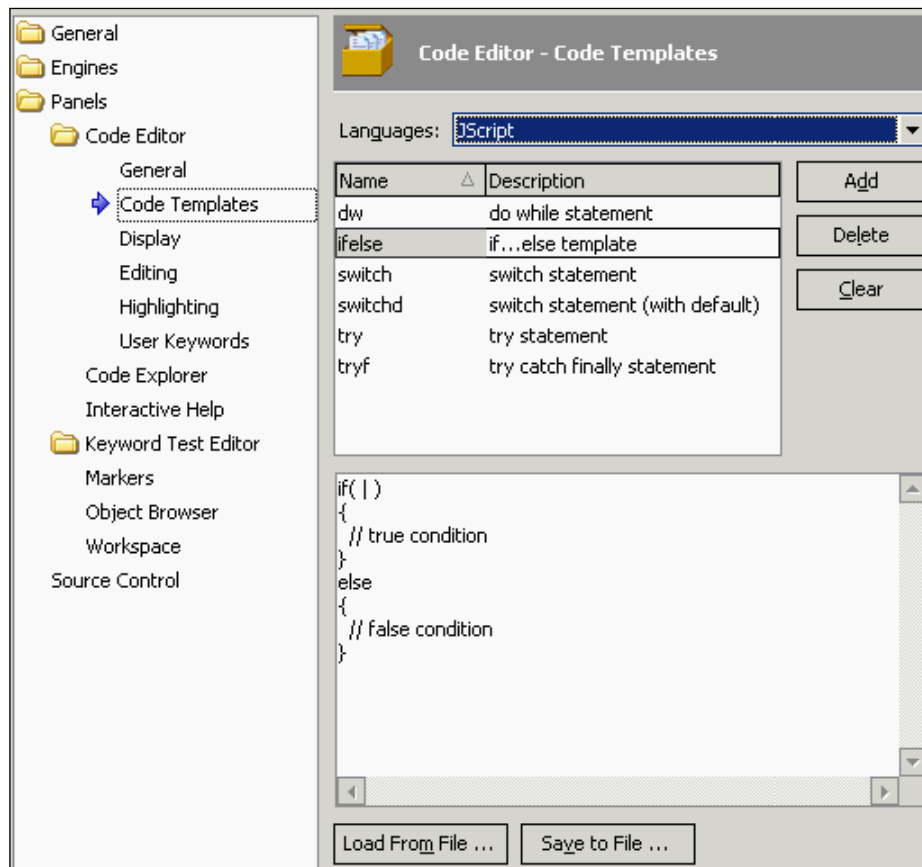
6. Click on the **OK** button.
7. Now, proceed to open any unit and press the following keys combination: `Ctrl + J`. You will see a listing of available templates on the screen.
8. Select the newly created template `ifelse` to witness exactly the same code appear in the editor as previously inputted in the listing of the templates.

## How it works...

We have added a new code template for the `if...else` construction for JScript language, and it can now be used in any TestComplete project.

Please note that code templates are different for all languages, so if you want to add the same construction for C++Script language, you need to repeat the preceding steps again.

In the following screenshot you can see how our template will look after performing all these steps:



## There's more...

Just as TestComplete settings, code templates are possible to save onto the file on a HDD to be used afterwards on another computer. It can be done using the **Save to File** and **Load From File** buttons in the templates' settings panel.

## Delaying script execution

Sometimes, there is a need to suspend script execution for some time to allow for synchronization with the work of the application. In this recipe, we will deal with the simplest method to provide such a timeout.

### How to do it...

In order to demonstrate the delay, we will use the `Test1Modified2` function from the *Modifying the recorded test* recipe:

1. Modify the `Test1Modified2` function by adding one line into it:

```
for(var i = 0; i < aButtons.length; i++)  
{  
    wCalcPlus.Window("Button", aButtons[i]).ClickButton();  
    aqUtils.Delay(2000);  
}
```

2. If you launch the function `Test1Modified2`, it will be apparent that upon each Calculator Plus button-click `TestComplete` will delay execution of the script at the rate of 2 seconds, and then continues with the flow of execution.

### How it works...

The `aqUtils.Delay` method pauses script execution for a specific range of milliseconds (for example, 2000 milliseconds equals 2 seconds).

This delay does not account for any factors (for example, CPU speed of the computer or network connectivity speed), which means one and the same delay can be too large or too small in different conditions. In first case, we will continually have errors in the log; while in the second case, scripts will have a useless standstill over the given period of time.



This is why it is not recommended to use this method too often as it's not reliable enough. Instead, it is better to use the `Wait` method that observes the moment an event is triggered.

## There's more...

There's one case when usage of the method `aqUtils.Delay` method is mandatory: when we expect a specific event in the loop to be triggered. Let's say we need to wait for the creation of the `c:\somefile.txt` file. In this case, the code will be as follows:

```
while(!aqFile.Exists("c:\\somefile.txt"))
{
    aqUtils.Delay(500, "Waiting for file...");
}
```

If the delay is not added to this code, `TestComplete` will keep checking too often, driving the CPU usage up to 100 percent, and thereby significantly slowing down work of other applications.

## See also

- ▶ You can learn more about synchronization in scripts in the *Waiting for an object to appear* and *Waiting for a property value* recipes in *Chapter 5, Accessing Windows, Controls, and Properties*

## Adding and removing project items

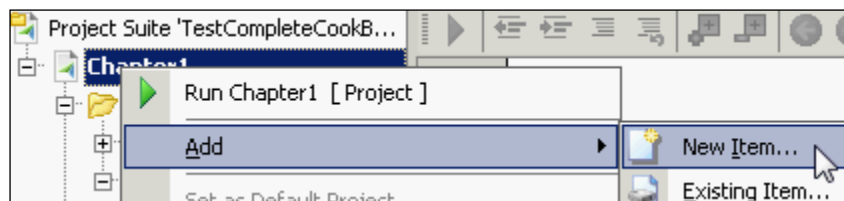
In `TestComplete`, a lot of features are implemented as project items, which implies the need to add a corresponding element to the project prior to using "this or that" functionality.

In this recipe, we will look into an example for adding a **Manual Tests** element to the project (the element is meant to store the tests that are necessary to run manually). Additions of other elements to the project are done similarly.

## How to do it...

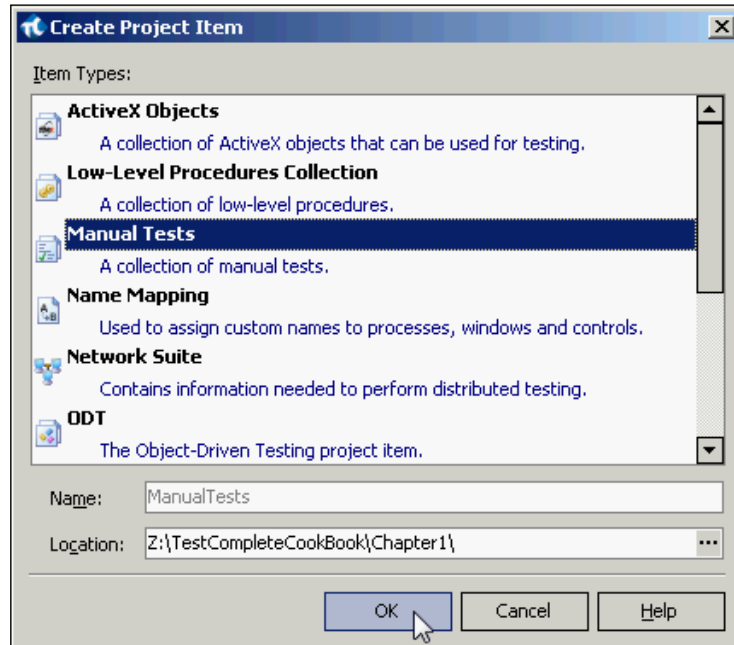
In order to add a project item, we need to perform the following steps:

1. Right-click on the name of the project in the **Project Explorer** panel and select the **Add | New Item** menu item:





2. In the opened **Create Project Item** dialog window, select the necessary element (**Manual Tests** in our case):



3. Click on **OK**.  
As a result there will appear in our project a new element **Manual Tests**, which is workable as if from within TestComplete as well as directly from the scripts (if such a possibility is in store for the element in view).
4. To delete the project item from the project, right-click on it and select the **Remove** menu item.
5. In the opened window, select one of the removal methods: **Remove** or **Delete**.

### How it works...

It is necessary to use the same action (**Add | New Item**) for adding elements to the project and for addition of other sibling elements. For example, to add a new test to the **Manual Tests** group, right-click on the **Manual Tests** element and go for the **Add | New Item** menu item.

- ▶ The **Remove** operation removes only the element of the project, while the element itself remains on your HDD. Later on, it could be added back into the project by selecting **Add | Existing Item** from the contextual menu of the project.
- ▶ The **Delete** operation will remove the elements of the project and all the project-related files.

## There's more...

Support of different components (.NET, Java, Win API, and others) as well as of third-party controls (Infragistics, DevExpress, and others) is implemented in TestComplete with the help of extensions. To look up the list of supported and included extensions, and to disable those which you don't use, opt for the **File | Install Extension** menu item and uncheck the unnecessary ones.

## Understanding how TestComplete interacts with tested applications

In this recipe we will deal with TestComplete workings with various windows and control elements.

## Getting ready

Launch the Calculator Plus application (C:\Program Files\Microsoft Calculator Plus\CalcPlus.exe) and make sure it is in Standard mode (the **View | Standard** menu item is checked).

## How to do it...

In order to learn how TestComplete works with tested applications we need to perform the following steps:

1. Let's make a simple test and run it.

```
function Test2()
{
    var pCalc = Sys.Process("CalcPlus");
    var wCalc = pCalc.Window("SciCalc", "Calculator Plus", 1);
    wCalc.Activate();
    wCalc.Window("Button", "2").Click();
    wCalc.Window("Button", "+").Click();
    wCalc.Window("Button", "2").Click();
    wCalc.Window("Button", "=").Click();
    var result = wCalc.Window("Edit", "").wText;
    Log.Message(result);
}
```

2. In the result of the calculus, the log will have several messages on the buttons that have been clicked and the result of the add-up equation 2+2 from the text field.

## How it works...

In the first line, we initialize the `pCalc` variable and assign it with the object that is returnable by the `Process` method. The `Process` method takes two parameters: the name of the process and its index. By default, the index is equal to 1 and can be omitted.

In the next line, we initialize a new variable `wCalc`, while using the previously created variable `pCalc` this time around.

The `wCalc` variable will have a value assigned that is returned by the `Window` method. This method takes in three parameters:

- ▶ `WndClass`: This parameter specifies the window class. The value of this property is viewable in **Object Browser** (the `WndClass` property).
- ▶ `WndCaption`: This parameter specifies a heading of the window, as seen by the user. The value of this property is also viewable in **Object Browser**.
- ▶ `GroupIndex`: This parameter specifies the current state of the window among the other windows of the same class (the so called, Z ordering). The `GroupIndex` parameter can also be omitted in case of unique control.

In the four lines to follow, we are working with the control elements of the type `Button`, while using the same `Window` method.

In the last line but one, we are working with a new control element `Edit`, by respectively signifying such a class for it. Since this element has no heading at all, the second parameter is an empty string.

In this same string, we go about creating a new variable `result` and assigning it with a value of the `wText` property of the text field—in order to have this value outputted to the log.



Please note that access to any control element, regardless of its class and level of hierarchy, is carried out with the help of the `Window` method! This is true only for Win32 applications.

If you do not know which class or heading of the necessary control element is to be taken up, make use of the **Object Spy** utility and look up the corresponding properties of the control element of choice.

## There's more...

In our examples we are handling an ordinary Win32 application.

Apart from this, there is a good deal of other types of applications (.NET, Java, Delphi, and so on), and for each type there are proprietary methods of access and control. For example, for .NET applications it is the `WinFormsObject` method, for Delphi applications it is the `VCLObject` method, and so on.

Hence, if you have, say, a control element, that is identifiable by TestComplete as

```
Sys.Process("myapp").WinFormsObject("DotNetWinClass", "App Caption")
```

you will not be able to address it with the help of the `Window` method:

```
Sys.Process("myapp").Window("DotNetWinClass", "App Caption") // <=
WRONG!
```

This is incorrect and will not work! For each type of application, correct methods should be used (they can be looked up in **Object Browser**).

## See also

- ▶ Usage of the `Log.Message` method is considered in greater detail in *Chapter 6, Logging Capabilities*, that is dedicated to working with the log

## Understanding Object Browser

In this recipe we will get familiar with an important TestComplete tool – namely, the **Object Browser** panel.

The **Object Browser** panel represents all the processes and objects visible for TestComplete in the system, which are workable from within TestComplete.

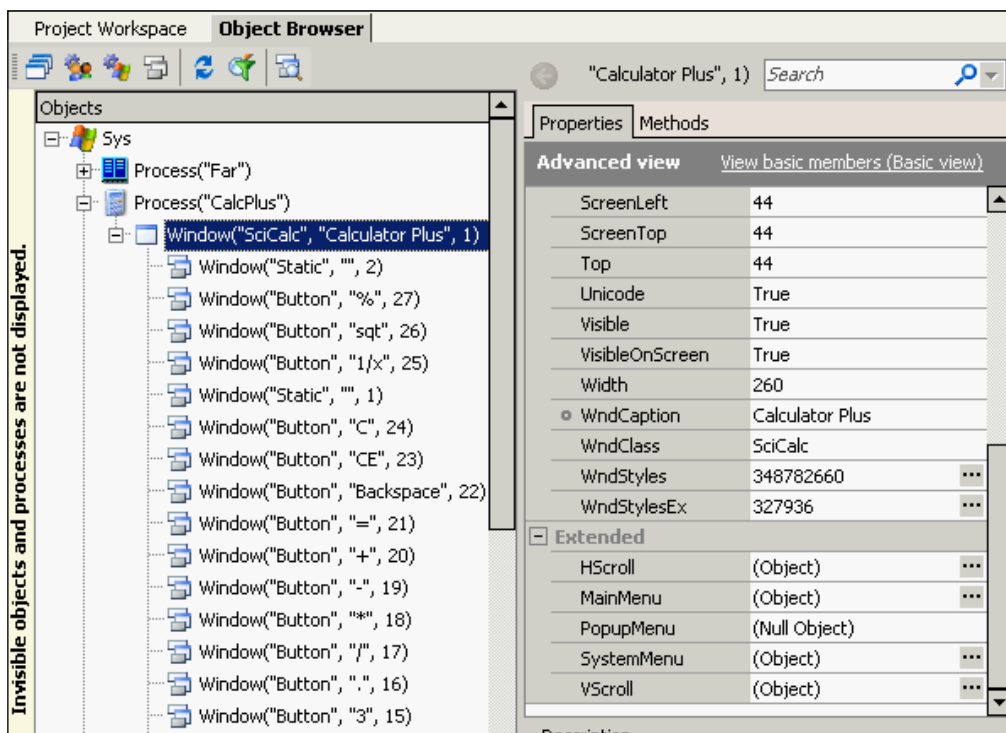
## Getting ready

We will need to run Calculator Plus as a target application to learn on (C:\Program Files\Microsoft Calculator Plus\CalcPlus.exe).

## How to do it...

In order to get acquainted with **Object Browser** we need to perform several steps:

1. Click on the **Object Browser** tab in the left part of the TestComplete window (near the **Project Workspace** pane) or press the following key combination: *Ctrl + Alt + O*.
2. In the collapsible list, expand the node **Sys**, and then the **Process ("CalcPlus")** node.
3. In the opened node, click on **Window("SciCalc", "Calculator Plus", 1)**.
4. On the right-hand panel click on the link **View more members (Advanced view)**:



## How it works...

The **Object Browser** tab displays all the system nodes within immediate access. On the **Objects** panel in the left-hand part of the collapsible list, all the system objects are shown. The root node **Sys** is the main object in the given hierarchy; it is particularly instrumental for accessing all the system nodes.

Sibling nodes of the **Sys** object are the processes. Each process, in its turn, has child elements, the windows. Also, the main window has child elements, the controls, with which we are working (buttons, data input, lists, toolbars, and others).

Depending on the application type, its complexity, and the customizable settings of the project, this hierarchy may be quite sophisticated. The matter is that TestComplete represents all the application elements, even invisible ones (which are used for positioning controls).

Each item of the **Objects** tree is an object itself (including the **Sys** object) with various properties and methods. By default, a short-list view is enabled for the displayed properties. By clicking on the **View more members (Advanced view)** link we have displayed a complete list without any exceptions.

Properties and methods of the selected item are displayed on the corresponding tabs in the right-hand part of the TestComplete window. Every element has several standard properties always available (such as width, ID, and class); also, the extended properties may be available.

The majority of properties are read-only; however, some of them can be accessed in read-write mode to have them changed. If that's the case, we will see a small circle near the name of the property (for example, the `WndCaption` property of the main Calculator Plus window). Such properties can be modified from the scripts directly.

Apart from simple properties there are compound properties with values as objects. In this case, in the property-value field, the value of **(Object)** will be signified. In order to view the value of such property, it is necessary to double-click on its name; this will cause the list to update by displaying properties and methods of the viewed compound property.

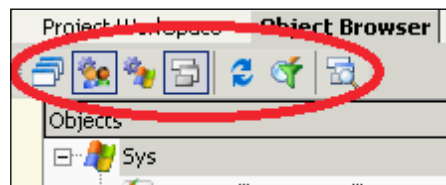
For example, double-clicking on the **MainMenu** property, we will see a list from two properties: the number of child elements and the menu items.

In this example, the **Items** property is also a compound one, meanwhile this is a property with parameters. Clicking on the **Params** button, we can have either the index of the menu item assigned or its caption and once more the compound property that corresponds the menu item.

In the process of work (especially, with complex control items), we will be working with **Object Browser** in order to locate the necessary properties.

### There's more...

In the topmost part of the **Object Browser** panel above the **Objects** tree, there are several buttons which make it possible to filter the displayed, processed, and other objects. For example, it is possible to hide invisible objects, system processes, and so on; and even selectively display the processes with the help of the filter.



The fewer processes and objects displayed on the **Objects** panel, the faster the information is updated for all the objects in view.

We recommend usage of the **Show Tested Applications Only** mode when it is possible. If necessary, one can connect, with the help of the filter window, only those processes with which you are currently working.

## See also

- ▶ Tested applications are considered in greater detail in the next chapter.
- ▶ If you are unable to find necessary control within the **Object Browser** tree, read the *Using Object Spy* recipe.
- ▶ The complexity of object hierarchy may be changed by modifying the **Object Tree Model** option. The *Choosing Object Tree Model* recipe in *Chapter 5, Accessing Windows, Controls, and Properties*, will help you to learn more about it.
- ▶ The following two recipes will give even more useful information on controls and windows:
  - *The Understanding how TestComplete interacts with tested applications* recipe
  - *The Understanding the window's life cycle* recipe in *Chapter 5, Accessing Windows, Controls, and Properties*

## Using Object Spy

If the tested application is a quite complex one and contains many different controls, locating the necessary element in the **Object Browser** panel may be quite a challenge.

To facilitate the task at hand, we can use the **Object Spy** utility.

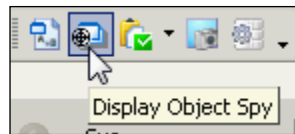


If you are using TestComplete of Version 7 or below, Object Spy will go by the name of **Object Properties**, and is no different otherwise from Object Spy.

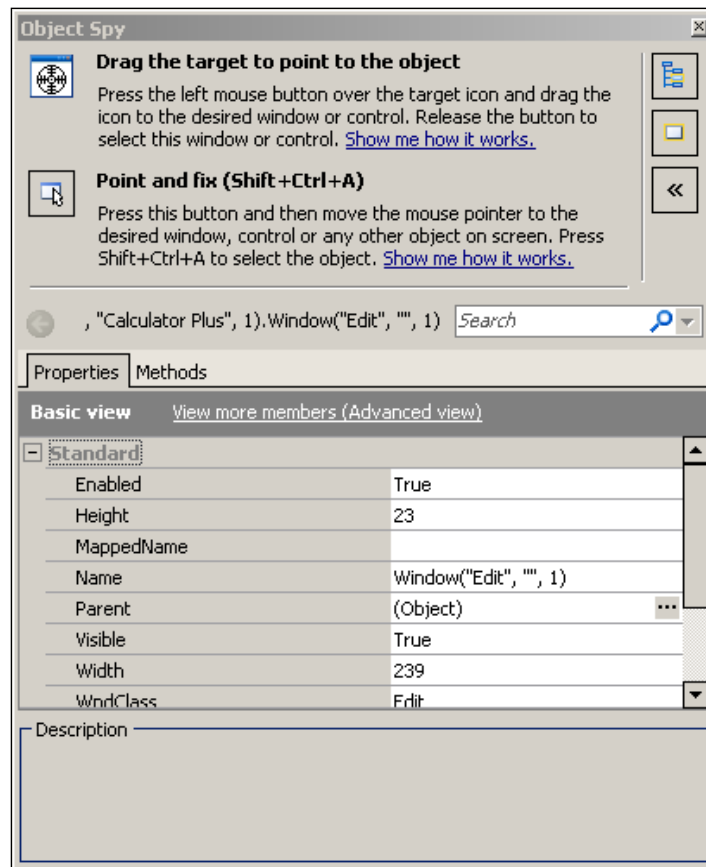
## How to do it...

In order to get acquainted with Object Spy we need to perform several steps:

1. Click on the **Display Object Spy** button on the toolbar.



2. In the result, the TestComplete window will minimize and the screen will have the **Object Spy** window displayed.



3. Now we need to signify the particularly necessary object in the **Object Spy** window. To this end, we have the following two methods:
  - Drag-and-drop by the mouse the sign of the target onto the necessary object and wait until the control element highlights it in red. Once this has occurred let go of the mouse button.
  - Press the **Point and fix** icon, hover the mouse cursor over the necessary object and press the following combination of keys: *Shift + Ctrl + A*.



In any case, in the list below, we will see a list of available properties similar to the list in **Object Browser**.

4. In order to see the selected element in **Object Browser**, click on the **Highlight Object** button in the **Objects** tree in the top-right corner of the **Object Spy** window.

### How it works...

TestComplete highlights the control elements, which it can identify. Do not be surprised if you happen to find out that some of the elements (for example, menu items, toolbar buttons, individual table elements, and so on) are not highlighted, while the whole group of elements is. Such behavior stands for either of the two:

- ▶ This is a compound control element and TestComplete can't highlight its internal component, only a whole compound control can be recognized and highlighted
- ▶ A given control element is not supported by TestComplete altogether

The **Object Spy** window displays information about controls the same way as **Object Browser** does. There is a tab control with available properties and methods at the bottom and a full name of the control above it. There is also a **Search** field, which allows us to quickly apply a filter to the list of properties and methods to find the necessary item in the list.

### See also

- ▶ The *Working with nonstandard controls recipe* in *Chapter 3, Scripting*, will guide you on working with nonstandard controls which are not recognized by TestComplete
- ▶ If you want to know how to find specific control by coded script, you can move to the *Finding objects by properties' values recipe* in *Chapter 5, Accessing Windows, Controls, and Properties*

# 2

## Working with Tested Applications

In this chapter we will cover the following recipes:

- ▶ Adding a tested application project item
- ▶ Running a tested application from the script
- ▶ Terminating a tested application
- ▶ Killing several instances of a tested application
- ▶ Closing a tested application
- ▶ Running a tested application in the Debug mode
- ▶ Running a tested application under a different user account
- ▶ Changing tested application settings dynamically from the script
- ▶ Running external programs and DOS commands
- ▶ Testing installers – running an MSI file

### Introduction

TestComplete can be used for testing different types of applications: desktop, console, web, and web services. In all cases we need to be able to run applications under test in different ways (with different command-line parameters or under different user's accounts), terminate them, and analyze the application's state.

In this chapter we will learn how to work with tested applications in TestComplete. In most cases, a special project item **TestedApps** is used for this purpose; however, in some cases other approaches may be of better use.

## Adding a tested application project item

Before getting down to the tested applications, it is necessary to add a corresponding project item to the project and come to an understanding of its parameters and settings.

### Getting ready

Any project can contain only one tested application project item, therefore prior to performing the following steps, make sure that the **TestedApps** element doesn't exist in the project. Otherwise, remove it by right-clicking on its name and selecting the **Remove** menu item.

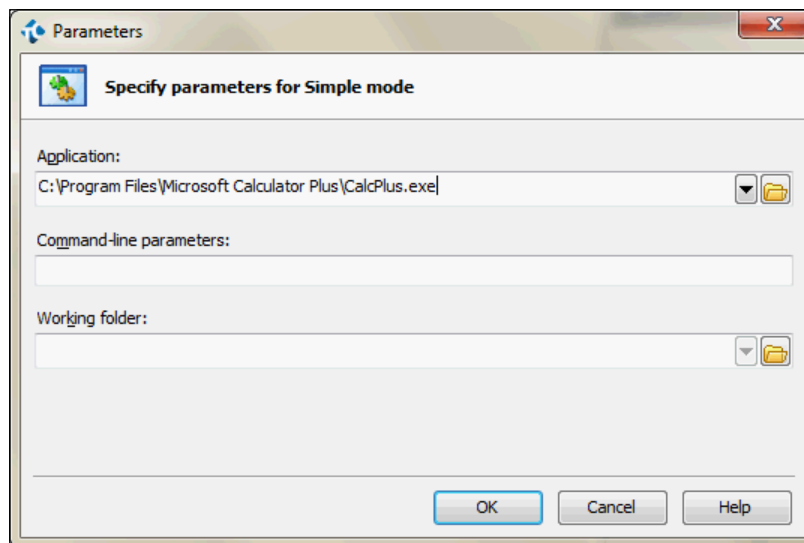
### How to do it...

For adding tested application project item perform the steps given as follows:

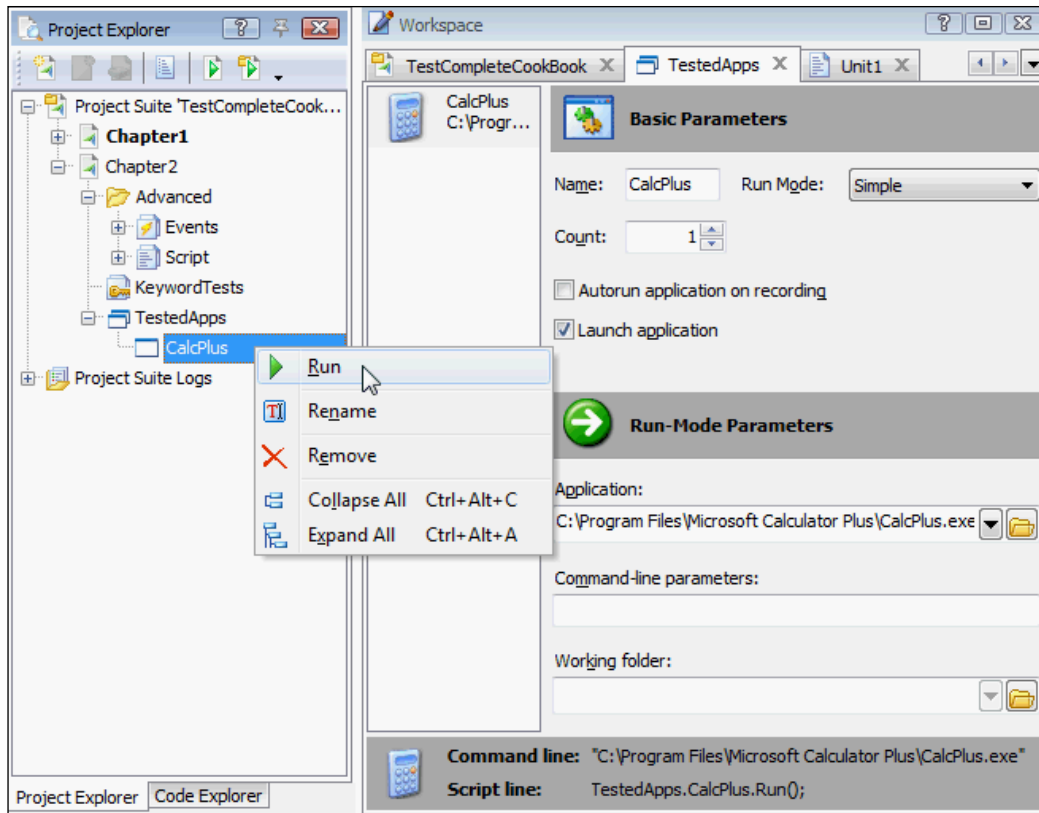
1. Right-click on the name of the project and opt for the **Add | New Item** menu item.
2. In the **Create Project Item** window select the **Tested Applications** element and click on the **OK** button.

There will appear a new **TestedApps** element in the project.

3. Right-click on the **TestedApps** element and opt for the **Add | New Item** menu item.
4. In the **Parameters** window enter the full path to the executable file (in our case, it is C:\Program Files\Microsoft Calculator Plus\CalcPlus.exe) in the **Application** field or click on the button next to the **Application** field and locate this file with the help of the **Select Tested Application** window.



5. Click on the **OK** button.
6. Double-click on the **TestedApps** element in the project. On the **Workspace** panel there will appear a list of tested applications.



7. To launch the added application, right-click on the **CalcPlus** element in the project tree and opt for the **Run** menu item.

### How it works...

We can add as many applications to **TestedApps** as we see fit to work in TestComplete, and even several clones of the same application. For example, if you need to launch an application with different parameters, it is possible to add several copies of the same application and uncheck the **Launch Application** option for those clones that are not necessary at the moment.

## There's more...

All the parameters of the tested application (for example, **Name**, **Run Mode**, **Working folder**, and so forth) can be changed at any given moment of time, and not only in the **TestedApps** window, but also via the scripts directly.

If you need to launch several clones of the application, change the **Count** parameter for the tested application by signifying the necessary number of launchable copies in it.

In order to launch all the applications from the list, right-click on the **TestedApps** element and opt for the **Run All** menu item.

## See also

- ▶ If you want to learn more about tested applications in TestComplete, please visit the following link on the SmartBear website:  
<http://support.smartbear.com/viewarticle/31408/>
- ▶ More information about running tested applications can be found in the following recipes:
  - *The Running a tested application from the script recipe*
  - *The Changing tested application settings dynamically from the script recipe*

## Running a tested application from the script

In most cases we will need to run tested applications automatically from the test scripts, not manually from TestComplete IDE.

In this recipe we will learn how to launch tested applications from the script.

## Getting ready

Add the Calculator Plus and Notepad applications to the application project (right-click on **TestedApps**, and select **Add | New Item**). These applications can be located through the following paths:

- ▶ C:\Program Files\Microsoft Calculator Plus\CalcPlus.exe
- ▶ C:\Windows\notepad.exe



Pay attention, in our book we use Windows 7 32-bit machine for our example. The path to `CalcPlus.exe` on 64-bit machine will be different (`C:\Program Files (x86)\Microsoft Calculator Plus\CalcPlus.exe`)!

## How to do it...



In order to run a tested application we need to perform the following steps:

1. Create the following function and launch it:

```
function testRunApps()
{
    var pCalc = TestedApps.CalcPlus.Run();
    var pNotepad = TestedApps.notepad.Run();
}
```

This code will run both tested applications using the `Run` method.

2. In the result, both of the applications will launch, and the log will have the following messages written on the launched applications:

Type	Message
	The application "C:\Program Files\Microsoft Calculator Plus\CalcPlus.exe" started.
	The application "C:\Windows\notepad.exe" started.

## How it works...

Each tested application item has the `Run` method, which is used for running applications under test. With help of this method, the application is launched in the mode specified by its **Run Mode** property.

The `Run` method returns the process object of the launched application, which can be used further in script for interaction with the application under test.

If for some reason run attempt fails, an error will be generated: **Unable to run "<PATH\_TO\_THE\_APPLICATION>"**.

## There's more...

If you have several applications in the project, it is not necessary to launch each one of them one by one, as it is possible to launch all of them at once with the help of the `RunAll` method:

```
TestedApps.RunAll();
```

## See also

- ▶ In some cases there is no need to add an application to the **TestedApps** object to run it (for instance, to run a DOS command). The *Running external programs and DOS commands* recipe explains another approach to run applications.

## Terminating a tested application

Sometimes it becomes necessary to quickly terminate an application (for example, if we don't care about saving application results or the application stopped responding).

In this recipe we will learn how to forcibly close the application using a script.

## Getting ready

Add the Notepad application to **TestedApps**.

## How to do it...

In order to terminate a tested application we need to perform the following steps:

1. Create and launch the following function:

```
function testRunApps ()
{
    TestedApps.notepad.Run ();
    TestedApps.notepad.Terminate ();
}
```

2. In the result of the function call, the Notepad application will be launched and then closed.

## How it works...

The `Terminate` method will forcibly kill the process of the application, without giving it a possibility to perform standard actions that are executable in the normal termination mode (for example, when a user presses the termination button to close the program). This action is analogous to terminating the process thread in the Task Manager.

This is why we should use the `Terminate` method only when we are sure that it will not lead to data loss or application intactness.

Another reason to apply the `Terminate` method is validation of the tested application's behavior in case of emergency shutdown (for example, will it notify us of its shutdown upon the next startup, will it restore the files that were being edited on the verge of the termination, and so on).

Usage of this method for normal termination of the program is not recommended.

## See also

- ▶ If you need to terminate all instances of the application, the *Killing several instances of a tested application* recipe will help you
- ▶ The better way of closing applications under test is explained in the *Closing a tested application* recipe

## Killing several instances of a tested application

Sometimes, there arises a necessity to terminate several processes under one and the same name, while the number of open processes may vary; hence, the exact number of the processes to terminate remains unknown.

## Getting ready

Add the Calculator Plus application to **TestedApps** and launch its copies (three or more).



## How to do it...

In order to terminate several instances of the tested application we need to perform the following steps:

1. Launch the following function:

```
function testTerminateProcesses()  
{  
    var procName = "CalcPlus";  
    while (Sys.WaitProcess(procName, 10).Exists)  
    {  
        Sys.Process(procName).Terminate();  
        aqUtils.Delay(500);  
    }  
}
```

2. In the result of the function call, all the instances of Calculator Plus application will be terminated.

## How it works...

Since we do not know the exact number of launched processes, we resort to the loop, in which, with the help of the `WaitProcess` method, we check against the availability of any of the sought processes. If such a process is located, we then terminate it with the `Terminate` method and wait the half a second (500 milliseconds) that might be necessary for the termination to complete. Then, we check again for any outstanding processes running under the same name.

Exiting the loop takes place when there is no process under the given name.

## There's more...

We can make this function more universal by declaring `procName` as a parameter (and removing declaration of the variable in the very beginning of the function):

```
function terminateProcesses(procName)
```

Now we can call this function for termination of any of the processes, not just of the Calculator Plus termination, for example:

```
terminateProcesses("notepad");
```

If you need to close only one process, provided you know its index, it is possible to do it the following way:

```
Sys.Process("CalcPlus", 2).Terminate();
```

Here, 2 stands for the index of the process.

## See also

- ▶ Here we used the `WaitProcess` method for verifying the process' existence. The `Wait` methods are explained in details in the *Waiting for an object to appear* recipe in *Chapter 5, Accessing Windows, Controls, and Properties*.

## Closing a tested application

In the *Terminating a tested application* recipe we have come to understand how to forcibly terminate the tested application, as well as ascertained that this method suits emergency cases exclusively.

In this recipe at hand we will take up an example of application termination.

## Getting ready

Add a standard Notepad application to **TestedApps** (right-click on the **TestedApps** node in the project, then select **Add | New Item** and choose the `C:\Windows\notepad.exe` file).

## How to do it...

In order to close a tested application we need to perform the following steps:

1. Create and launch the following function:

```
function testCloseNotepad()
{
    var pNotepad = TestedApps.notepad.Run();
    var wNotepad = pNotepad.Window("Notepad", "Untitled -
        Notepad");
    wNotepad.Activate();
    wNotepad.Keys("Some text to prevent closing");
    TestedApps.notepad.Close();
}
```

2. In the result, the Notepad application will remain running, and in the log we will spot the following notification:

**The application "C:\Windows\notepad.exe" got a command to close, but it is still running, though the default timeout has expired.**

3. Now, modify the function by adding the following code to it, as shown in this example:

```
function testCloseNotepad()
{
    var pNotepad = TestedApps.notepad.Run();
    var wNotepad = pNotepad.Window("Notepad", "Untitled - Notepad");
    wNotepad.Activate();
    wNotepad.Keys("Some text to prevent closing");
    TestedApps.notepad.Close();
    if (pNotepad.WaitWindow("#32770", "Notepad", -1, 1000).Exists)
    {
        pNotepad.Window("#32770", "Notepad").Keys("~n");
    }
}
```

4. Now the Notepad application has closed; however, the notification is still there in the log. In order to get rid of the notification, let's replace the following line:

```
TestedApps.notepad.Close();
```

With this one:

```
wNotepad.Close();
```

Then launch the function again.

5. In the result, the Notepad application is closed and there are no notifications in the log about it.

## How it works...

When we close an application via the **TestedApps** object (as we have done in the first two examples), **TestComplete** prompts the application with a termination command and awaits for the application to complete. If this does not happen (as in our case, when Notepad awaits for us to push the button), **TestComplete** notifies us about that in the log.

Further on, we add the code, which verifies with the help of the `WaitWindow` method, whether the message box has appeared, and if so, we close it by pressing the buttons combination: *Alt + N* (which corresponds with pressing the **Don't Save** button). This way, we achieve the correct termination of the application.

And, finally, we substitute the call of the `Close` method for **TestedApps** with analogous one of the window. Thereby, the termination command is sent to the window and **TestComplete** does not wait for the application to close down.

## There's more...

To close the application, it is not necessary to call the `Close` method. Instead, one could apply other methods for termination, for example, opting for the **File | Exit** menu item or sending such a combination of keys as `Alt + F4` with the help of `Keys` method. Note, however, that sending key sequence requires that the target window be active, otherwise, it is possible to send the sequence to another window.

## See also

- ▶ If you are still unsure how the `WaitWindow` method works in the previous example, read the *Waiting for an object to appear* recipe in *Chapter 5, Accessing Windows, Controls, and Properties*

## Running a tested application in the Debug mode

Sometimes, at the point of an application launch, we need to know the details like which modules are being loaded by the application in view. To this end, TestComplete has an option for launching a debugger mode.

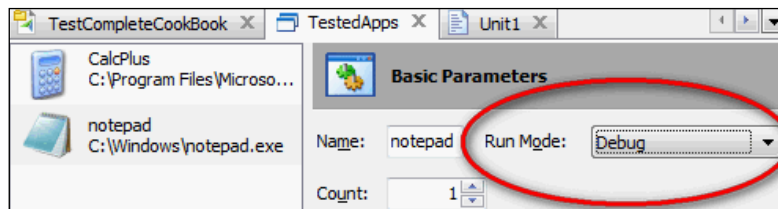
## Getting ready

Add a standard Notepad application in **TestedApps**.

## How to do it...















In order to run a tested application in Debug we need to perform the following steps:

1. Double-click on the **TestedApps** project's element.
2. In the open **TestedApps** pane go for Notepad.
3. Set the **Run Mode** option to the **Debug** value.



4. Create a new function with a single code line `TestedApps.notepad.Run()` and execute it.

In the result, as shown in the following screenshot, the log will have the information about all the files that are being uploaded by the application at the point of launch:

Type	Message
	Module Unloaded: C:\Program Files\SmartBear\TestComplete 9\Bin\...
	Thread Create ID: 5700
	Thread Exit ID: 5700 Exit Code: 0
	Thread Create ID: 4840
	Module Loaded: C:\Program Files\SmartBear\TestComplete 9\Bin\Ex...
	Module Loaded: C:\Program Files\SmartBear\TestComplete 9\Bin\Ex...
	Module Unloaded: C:\Program Files\SmartBear\TestComplete 9\Bin\...
	Thread Exit ID: 2808 Exit Code: 0
	Thread Exit ID: 4840 Exit Code: 0
	Thread Exit ID: 2740 Exit Code: 0
	Thread Exit ID: 2700 Exit Code: 0
	Thread Exit ID: 444 Exit Code: 0
	Thread Exit ID: 5812 Exit Code: 0
	The application "C:\Windows\notepad.exe" started.

### How it works...

When we start the application in the **Debug** mode, TestComplete takes over as debugger for the application and keeps track of all the modules that have been uploaded by the application, all the emergent exceptions, and so on.

It is also possible to view detailed information for each individual message. To do so, one should click on the message in the log and switch to the **Additional Info** tab (under the log messages).

Meanwhile, a vast amount of information is entered to the log, thereby making it cluttered, this is why it is recommended that the **Debug** option should be used only in case of need (for example, if the application triggers fallback closing at the point of being launched, due to some unknown reasons), in other cases a simple mode is used.

Besides, in the **Debug** mode, the application works slower than in the simple ordinary mode.

## Running a tested application under a different user account

By default, TestComplete launches tested applications under the same user account under which TestComplete itself had been launched. At other times, however, it stands to reason one should be able to verify workability of the application under a different account name(s), for example, one with restricted permissions, and so on. To this end, the **Run As** option of the tested application has been specifically earmarked.

### Getting ready

Create a new user in the system:

1. Right-click on the **Computer** element in the main menu and opt for the **Manage** menu item.
2. Expand the **System Tools | Local Users and Groups | Users** element.
3. In the main menu, go for the **Action | New User** option.
4. In the open **New User** window, enter a user name (for example, `user1`), a password and disable the **User must change password at next logon** option.
5. Click on **Create**.

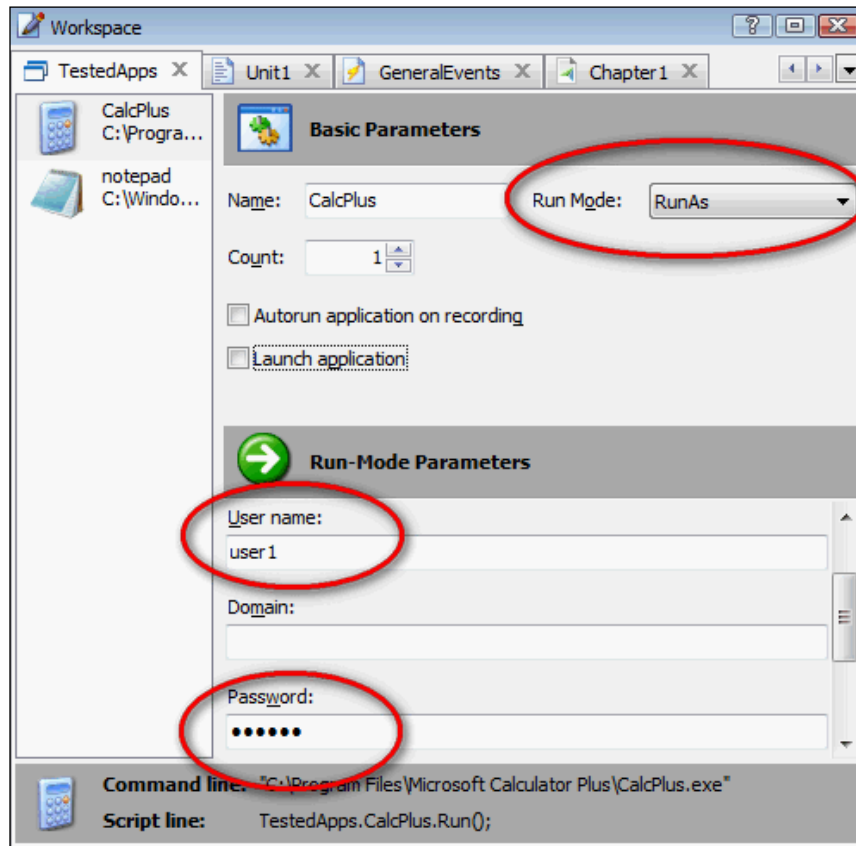
Add the Calculator Plus application to **TestedApps** and make sure it appears as a commonplace Windows application, as it's being launched (that is, the **View | Classic View** option has been enabled).

### How to do it...

In order to run a tested application as a different user we need to perform the following steps:

1. Double-click the **TestedApps** element and select the **Calculator Plus** in the right-hand pane.
2. In the **Basic Parameters** section, select the **Run As** element in the **Run Mode** drop-down list.

3. In the **Run-Mode Parameters** section, signify the name of the created user and the password.



4. Launch the application by either selecting the option **Run** from the contextual application menu in **TestedApps**, or by using the `TestedApps.CalcPlus.Run()` command in the script.
5. Calculator Plus will start up with the default view (not the **Classic View** option).

### How it works...

When we have changed the launching parameters of the calculator by setting the **Classic View** option, these customizations were saved for the current user. For a newly created user, it was the very first calculator application launch, and that's why default settings were applied. If we set the settings to **Classic View** once again (after calculator had been launched from under the `user1` account name), the settings would have to be saved to be accounted for upon subsequent launches of the application.

## There's more...

In order to launch the application from under another user, it is not entirely necessary to change application parameters, as one could simply apply the `RunAs` method of `TestedApps`. For example, in our example, the function call would look like this:

```
TestedApps.CalcPlus.RunAs("", "user1", "123456");
```

## Changing tested application settings dynamically from the script

In some cases, the path to the tested application may vary from one computer to another. For example, installing the 32-bit application onto the 32-bit system, application shall default to the `Program Files` folder; and in case of 64-bit system, programs end up in the `Program Files (x86)` folder.

Another example of using dynamic paths is testing different versions of the same application, when the application path also includes the version number.

We can simply create two copies of the tested application in **TestedApps** and toggle-switch each one on and off, depending on the system type. Otherwise, we could define the pathname dynamically and modify it (along the other parameters) prior to the launch.

## Getting ready

Add the Calculator Plus application to **TestedApps**.

## How to do it...

In order to change the tested application's path we need to perform the following steps:

1. The following example shows how we can change parameters of the application directly by scripting:

```
function testChangeAppSettings()
{
    if (Sys.OSInfo.Windows64bit)
    {
        TestedApps.CalcPlus.Path = "C:\\Program Files (x86)\\Microsoft
        Calculator Plus";
    }
    else
    {
```



```
    TestedApps.CalcPlus.Path = "C:\\Program Files\\Microsoft  
    Calculator Plus";  
  }  
  TestedApps.CalcPlus.Run();  
}
```

2. Launch it to verify that it is workable.

## How it works...

Here we define the bitness of the system with the help of the `System.OSInfo.Windows64bit` property and set the path to the tested application depending on the value of the passed variable.

In a similar manner, we can change other application parameters as well (name, parameters, and so on).



Before applying a certain property, read up on it in the reference book to make sure this property is not obsolete. Deprecated properties and methods are not recommended for use, as in the following versions of TestComplete they could be removed, for obsolescent properties and methods can be always replaced with more user-friendly alternatives.

## Running external programs and DOS commands

Sometimes, there arises a need to launch a program without adding it to **TestedApps**. This could be, as an example, a DOS command (`dir`, `del`, `copy`, and so on), one of the programs in use, or asynchronous launch of the program without awaiting its completion.

To do just that, we will apply the `WScript.Shell` object that is a standard component of the Windows Script Host.

We will consider three possible use cases for the launch: standard program launch, parameterized launch, and execution of the program from the DOS command prompt.

## Getting ready

On `C:\\` create the file with the name of `myfile.txt`.

## How to do it...

In order to run different commands we need to perform the following steps:

1. Create and launch the following function:

```
function runExternalCommand()
{
    var calc = "\"C:\\Program Files\\Microsoft Calculator Plus\\
CalcPlus.exe\"";
    var notepad = "C:\\Windows\\notepad.exe c:\\myfile.txt";
    var doscmd = "cmd /c copy c:\\myfile.txt c:\\myfilecopy.txt";

    var ws = Sys.OleObject("WScript.Shell");
    ws.Run(calc, SW_SHOWNORMAL);
    ws.Run(notepad, SW_MINIMIZE);
    ws.Run(doscmd);
}
```

2. In the result of the function call, the Calculator Plus and Notepad applications will be launched with the `myfile.txt` file opened; and the file `myfile.txt` will be copied to the `myfilecopy.txt`.

## How it works...

To launch the commands we applied the `Run` method of the `WScript.Shell` object. The first parameter that is passed to this method is the to-be-executed command per se. The second parameter is the mode of showing the window. We have launched the calculator in the standard mode, and the Notepad in the minimized mode. The second parameter is optional and can be omitted (as shown on the following example).

Please, pay attention to specificity of this method's workings:

- ▶ If the path to the file contains spaces, it has to be escaped with double quotes (for example, Calculator Plus instance). The same should be done if the spaces are encountered in the parameters.
- ▶ Parameters are passed in the same line of code together with the path of the launched file.
- ▶ To prompt DOS commands, we apply the `cmd` program with the parameter `/c`, which is a parameterized command that launches it and completes its session.

## There's more...

The `Run` method of the `WScript.Shell` object can also open files with the help of mapped (associated) applications. For example, opening the `myfile.txt` file is also doable as follows:

```
Sys.OleObject("WScript.Shell").Run("c:\\myfile.txt");
```

Another method to launch programs is through use of `WinAPI`. For example, you can launch Notepad using `WinAPI` like this:

```
Win32API.WinExec("C:\\Windows\\notepad.exe", SW_SHOWNORMAL);
```

## See also

- ▶ A good example of running external commands is running an MSI file, which is explained in the *Testing installers – running an MSI file* recipe

## Testing installers – running an MSI file

Launching MSI files is controlled by the `msiexec` program. When we launch MSI file, the `msiexec.exe` program is the one that's being launched for real, from the `C:\Windows\System32\` folder.

This is why, as in the case of launching the other file types (taken up in the previous recipe), we can launch MSI files in two of the possible ways: launch the MSI file directly or pass it as a parameter to the `msiexec` program.

## Getting ready

Place the Calculator Plus installer (the `CalcPlus.msi` file) to the root directory of `C:\\`.

## How to do it...

In order to run an MSI file we need to perform the following steps:

1. Launch the following function:

```
function testRunMSI()  
{  
    var msi1 = "C:\\CalcPlus.msi";  
    var msi2 = "msiexec.exe /i C:\\CalcPlus.msi";
```

```
var ws = Sys.OleObject("WScript.Shell");  
ws.Run(msi1);  
ws.Run(msi2);  
}
```

2. In the result, we will have two copies of the Calculator Plus installer launched.

### How it works...

The inner workings are the same as in the previous recipe; however, the only peculiarity is that, in order to launch the MSI file as a parameter of the `msiexec` program, we will have to attach the parameter `/i` to it, in order to signify the path to the installer, to be passed further on.

### There's more...

If the `msiexec` program were to be launched without parameters, it would display only the window with the list of usable options that are available. For example, the `/quiet` parameter allows launching silent mode installation, that is, without interacting with the user. This parameter is handy to use in case we need to install a new version of the tested application before running tests.

### See also

- ▶ Launching different applications and commands is discussed in detail in the *Running external programs and DOS commands* recipe



# 3

## Scripting

In this chapter we will cover the following recipes:

- ▶ Entering text into text fields
- ▶ Using wildcards to process objects with variable names
- ▶ Structuring code using loops
- ▶ Creating property checkpoints
- ▶ Creating object checkpoints
- ▶ Using global variables
- ▶ Testing multilingual applications
- ▶ Working with nonstandard controls
- ▶ Organizing script code in the project
- ▶ Handling exceptions
- ▶ Handling exceptions from a different unit
- ▶ Creating framework using the OOP approach

### Introduction

Recording in TestComplete allows us to fast create testing scripts, and recorded tests are easy to read and maintain. Scripting allows us to create powerful and maintainable tests. It also allows us to create handy testing frameworks and solve complicated tasks with programming. Scripting also requires good programming skills and general understanding of project's design. In this chapter, we will consider different programming tasks one may face when manually creating test scripts. We will use JScript programming language for our examples.

## Entering text into text fields

The major TestComplete workload consists of interacting with the tested application via text-input, button-click, selecting elements from the drop-down list, and so on. Of course, automated checks execution is the basic purpose of our tests; however, in order to carry out any checks whatsoever, one should begin with a number of actions, all of which can take much time to complete.

In this recipe, we will consider two methods to work with the controls elements, using text field as an example and decide when it's better to apply either method.

### Getting ready...

Launch the standard Notepad application (C:\Windows\notepad.exe).

### How to do it...

In order to review two methods of text inputting, we will create two functions `testEditControl1` and `testEditControl2`.

1. Create and launch the first function:

```
function testEditControl1()
{
    var pNotepad = Sys.Process("notepad");
    var wNotepad = pNotepad.Window("Notepad", "*");
    var tEdit = wNotepad.Window("Edit");
    var str = "let's try writing something here[Enter]in two
lines";

    wNotepad.Activate();
    tEdit.Keys("^a[Del]")
    tEdit.Keys(str);
}
```

2. The second function is as follows:

```
function testEditControl2()
{
    var pNotepad = Sys.Process("notepad");
    var wNotepad = pNotepad.Window("Notepad", "*");
    var tEdit = wNotepad.Window("Edit");
    var enterStr = String.fromCharCode(13) +
String.fromCharCode(10);
    var str = "let's try writing something else here" +
enterStr + "in two lines again";
    wNotepad.Activate();
    tEdit.wText = "";
    tEdit.wText = str;
}
```

3. After executing these two functions, the difference is clear: the speed of executing the second function is much faster than the first one, though the result is the same in both cases.

### How it works...

The first few lines of both of the preceding functions are the same. We simply declare the variables corresponding to the application objects (process, main window, and text field) that are to be used further. After that, we declare a variable of the `String` type, whose text we will enter to a Notepad input field (note that the values of the same variable are different in these functions. We will talk about this difference later in this recipe).

Further, we will activate the Notepad window and clear the text field, for starters (against any prior possibility of some text therein). Then we will proceed in inputting a new text. The procedure of clearing the field and inputting text is of an interest to us in particular.

In the first instance, we have applied the `Keys` method, which allows inputting text to any window or a controls element, as does the user.

In the second instance, we assign new values to the `wText` property directly, as seen in the example, and this method works faster than the `Keys` method.

This, however, does not mean one should always use the second method. To tell the truth, assigning values directly to a property is not a straightforward method, since users behave otherwise. This is just a method to expedite filling out the window data, and in some cases it is truly justifiable. For example, when working with Internet Explorer, the `Keys` method works quite slowly (due to Internet Explorer peculiarities, `TestComplete` has to forcibly decrease the inputting speed, otherwise a number of symbols will be omitted at the point of entry). This is why, when testing applications in Internet Explorer, it is best to apply the properties intrinsically.

### There's more...

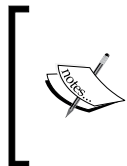
Another example, when using properties is preferable is when entering large amount of data. If we have many text fields having several text lines inputted into, the process will be a bottleneck. By assigning values directly to the properties, we can significantly bring our scripts execution up to speed.

However, along with its advantages, working directly with properties has some downsides. Specifically, when assigning values to properties directly, some events may misfire (for example, an event-handler that is bound onto inputted text to the text field, thus providing the auto-filling of the text). In simple cases, we will not be able to test such a function.



In sophisticated instances, we will be able to impact application performance and evoke some errors, otherwise not reproducible manually. This is why the thumb rule of applying properties instead of the `Keys` method is this: use the property-assigning method only in those cases when you are absolutely positive that this will not impact the application workability! It is best to discuss with the programmers, who are responsible for the application interface creation, whether or not it is acceptable to access properties directly.

Another complexity of the two methods lies in the fact they handle specific symbols differently. The `Keys` method transmutes some specific lines into button-click events. In our example, we are dealing with the `[Enter]` line, which is getting transmuted to pressing the `Enter` key at the point of text entry. If we are to try assigning this value directly to the property, the text field will have the `[Enter]` line appearing. This is why, in order to break over to the new line in the second function, we have to create a specific variable `enterStr` with the two symbols that correspond with returning the caret in Windows OS. Should we wish for the line to be outputted to the log with the help of the `Log.Message` method, we would have to replace the sequence with the symbol of `\n` (which corresponds to the line-break in JScript). Thus, if we would like to use one and the same string for both of the methods, we may come across some hardships which we would have to resolve additionally by replacing the symbols in the string, which can be grueling at times.



The names of the properties can be different for various controls elements. For instance, in our example, the text of the input field is stored in the property `wText`; however, for the other types of controls elements (.NET, Java, or simply own-drawn controls elements) this could be the `Text`, `Caption`, or any other property.

## Using wildcards to process objects with variable names

The majority of the controls elements have unchanging headers, texts, and so on (that is, properties by which `TestComplete` makes a difference). In some cases, these properties are dynamic, that means, they change depending on their conditions. For example, heading of the Notepad application starts with the name of the open file (or the word **Untitled** for a newly created file). Some URLs contain dynamic parts in parameters (for example, session identifier, which is not liable to change, but to be generated randomly). In such cases we apply symbols of the batch replacement (that is, the **wildcards**).

### Getting ready...

Create the file `myfile.txt` in the root directory of the disk `C:\` with any contents.

## How to do it...

In order to handle dynamic caption of the Notepad application we need to do the following:

1. Create and call the function that would open the Notepad application in dual modes: in the mode of a new file and in the mode of opening existing file, as shown in the following example:

```
function testWildcardsUsage()
{
    var notepad;
    var filename = "C:\\myfile.txt";
    var notepadCmd = "C:\\Windows\\notepad.exe";
    var ws = Sys.OleObject("WScript.Shell");

    ws.Run(notepadCmd);
    notepad = Sys.Process("notepad").Window("Notepad", "* - Notepad");
    Log.Message(notepad.WndCaption);
    notepad.Close();

    ws.Run(notepadCmd + " " + filename);
    notepad = Sys.Process("notepad").Window("Notepad", "* - Notepad");
    Log.Message(notepad.WndCaption);
    notepad.Close();
}
```

2. While the function is executing, Notepad will be opened twice and successfully recognized by TestComplete in both cases (the `notepad` variable is successfully initialized and used for closing Notepad); regardless of the fact that the heading of the file is different in either case.

## How it works...

In the first block of code, we declared and initialized a variable for further usage. In the two subsequent blocks of code, we will execute similar actions: opening the Notepad application, sending the header of the just-opened window to the log, and then closing the Notepad application.



These two recurring blocks of code are not a good example for programming style, because they contain one and the same code with an insignificant difference. We have used this approach only for the purpose of helping visualize the example, in real projects such an approach is banned from usage and should not be recommended.

There are only minute differences in these two blocks of code. In each case the headings of the Notepad window are different; however, we still have the possibility to use one and the same code to work with the window (this has been highlighted with the bold text). This is achieved by using a batch-replacement symbol \* (the asterisk). In TestComplete, we have two symbols for the batch-replacement:

- ▶ **\* (asterisk)**: This corresponds to any sequence of symbols (stands for several symbols, one symbol, or a lack of any symbol)
- ▶ **? (question mark)**: This stands for any singular symbol

We use such a form to address the Notepad window:

```
Window("Notepad", "* - Notepad")
```

We have notified in TestComplete that the window should pertain to the Notepad class, and the heading of the file should begin with any sequence of symbols and end with the line " - Notepad ". This is exactly why in both the cases we could leave the script code unchanged in order to proceed working with the window of the application. Batch-replacement symbols can be placed anywhere across the string of symbols (in the beginning, in the middle, or at the end) or they can crop up anywhere inside the string once or multiple times.

Besides, batch-replacement symbols can be usable in TestComplete almost everywhere, not just to cover the heading of the applications windows. For example, we could create description of the Notepad window as follows:

```
Window("*", "* - Notepad")
```

This would correspond to the window with any class with the heading that we have taken up so far. We should, however, abide by discretion not to use batch-replacement symbols just anywhere, as too generic makeup of the such strings can lead to conflicting matches and mismatches (when one and the same description of an object matches several real objects in the application).

## There's more...

The preceding example will not work for non-English system locales, because the name of the Notepad window will be different in the application caption. To solve this, we can also use wildcards: `Window("Notepad", "* - *")`. Here, we left the class `Window` untouched (as it is the same for all locales), but replaced the `Notepad` text in the caption with asterisk to match any word. If the text in the control contains asterisk, you can use double-asterisk to match the character. For instance, a Calculator application has multiplication button. We can address this button like this:

```
Window("Button", "**")
```

## See also

To know more about how to launch applications refer to the following recipes:

- ▶ The *Running external programs and DOS commands* recipe in *Chapter 2, Working with Tested Applications*
- ▶ The *Posting messages to the log* recipe in *Chapter 6, Logging Capabilities*
- ▶ The *Organizing script code in the project* recipe
- ▶ The *Finding objects by properties' values* recipe in *Chapter 5, Accessing Windows, Controls, and Properties*

## Structuring code using loops

In the *Modifying the recorded test* recipe in *Chapter 1, Getting Started*, we have briefly dwelled on the topic of loops, having used the `for` loop for clicking several buttons. In this recipe, we will consider this issue in greater detail. Testers who are not familiar with programming, would often make the error of redundantly repeating code. This may relate to recording tests via recorder or with copying blocks of code with further insignificant changes. This is considered to be bad style in programming, because, in the future, when we would need to introduce any changes into the code, we would have to change it in all the places where it had been copied to.

Let's consider such an assignment. We have to create a smoke-test for Notepad, which will be, first of all, quick, and secondly, will carry out only superficial checks just to make sure that the application does not have some critical drawbacks that could stand in the way of conducting complete testing. As a task for such a smoke-test, we will open all the available dialog windows that can be opened in the Notepad application, to check their headings, and then close them. Such windows in Notepad application are eight in number: **Open**, **Save As**, **Page Setup**, **Print**, **Find**, **Replace**, **Go To**, and **Font**.

The simplest way to create such a test would be just recording it. However, if in the future we will have new windows to be added to the test, we would have to copy the available code (several lines), change the headings of the menu items and headings of the windows and insert them at the end of the test. In the result, with each added window, our test will tend to bloat. If we make up our minds in the future to check availability of the **Cancel** button in the test on top of checking the heading of the window, we would have to re-write several lines of code and copy them as many times as many windows are set on testing. And this is just a smoke-test! If we are automating more complicated scenarios, the changes would be just as complex as well.

Let's take a look at how it would be possible to implement such a check correctly.

## Getting ready...

Create the file `myfile.txt` with any contents in the root directory of the `C:\` disk and start the Notepad application

## How to do it...

In order to structure code we need to perform the following steps:

1. First, let's write a script that will launch Notepad with the open file of `C:\myfile.txt`, select one of the necessary menu items (for example, **File | Open**) and close the dialog window that was opened at the point of selecting the underlying menu item. In the result we will obtain the following function:

```
function smokeTest()
{
    var pNotepad = Sys.Process("notepad");
    var wNotepad = pNotepad.Window("Notepad", "* - Notepad");
    wNotepad.MainMenu.Click("File|Open...");

    var dlg = pNotepad.WaitWindow("#32770", "Open", -1,
    3000);
    if(!dlg.Exists)
    {
        Log.Error("Dialog window didn't open");
    }
    else
    {
        dlg.Close();
    }
}
```

2. Now, it is necessary to modify the function in such a way that Notepad could open to the next file: `C:\myfile.txt`, and then go through several menu items and check whether the necessary window has been opened.
3. We have several menu items and several window headings that correspond to the given menu. The simplest way would be declaring two arrays: in the first array we would enumerate the menu items, and in the second array, the window headings.
4. After that in the loop we would iterate through the loop and select the necessary menu items one-by-one, checking into the corresponding window headings. The new function would look like this:

```
function smokeTest()
{
    var menus = ["File|Open...", "File|Save As...",
    "File|Page Setup...", "File|Print...",
```

```

"Edit|Find...", "Edit|Replace...",
"Edit|Go To...", "Format|Font..."];
var captions = ["Open", "Save As", "Page Setup",
"Print", "Find", "Replace",
"Go To Line", "Font"];

Win32API.WinExec("notepad.exe c:\\myfile.txt",
SW_NORMAL);
var pNotepad = Sys.Process("notepad");
var wNotepad = pNotepad.Window("Notepad", "* - Notepad");

for(var i = 0; i < menus.length; i++)
{
    wNotepad.MainMenu.Click(menus[i]);
    var dlg = pNotepad.WaitWindow("#32770", captions[i], -
1, 3000);

    if(!dlg.Exists)
    {
        Log.Error("Dialog window didn't open");
    }
    else
    {
        dlg.Close();
    }
}
}

```

5. This new code variation works in the loop with several windows. If, in the future, we have a new window to be added to the given test, we will simply add two new elements to the arrays `menus` and `caption`, while the script would already be aware how it should be handling things.

### How it works...

At first, we declare the two arrays which contain menu items and windows' headings. If new menu items are added to the tested application in the future, we will be able to add them for testing simply by adding new items to these arrays. Then we would launch the Notepad and initialize variables to work with the application.

Further on, in the loop, we go through all the elements of the menu (the `menus` array) and select the menu items one-by-one. Each time, having chosen the next menu item, we wait for the new window to appear with the help of the method `WaitWindow` lasting 3 seconds (3000 milliseconds). If the window does not pop up, the error notification is logged, if the window pops up, it is then closed.

## Creating property checkpoints

The pith and marrow of any testing lies in the execution of various kinds of checks. In this recipe we will come to grips with the simplest type of checks: **property checkpoint**.

This means that for a selected object we check the value of a certain property (for example, text, availability, visibility, and so on).

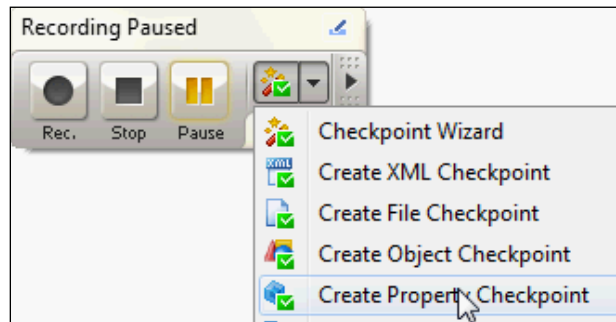
### Getting ready

Launch the Calculator Plus application (C:\Program Files\Microsoft Calculator Plus\CalcPlus.exe).

### How to do it...

In order to create a property checkpoint we need to perform the following steps:


1. Begin recording of the script (go to **Test | Record | Record Script** menu).
2. Switch to the **Calculator Plus** window and calculate some mathematical expression, for example, 2+7.
3. On the **Recording** pane, in the **Create New Property Checkpoint** drop-down menu, go for the **Create Property Checkpoint** option.




As a result, the **Create Property Checkpoint** window will show up on the screen.

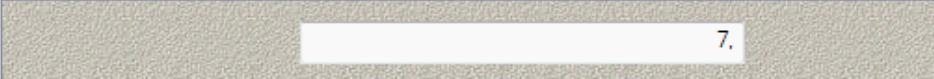
4. Drag-and-drop the target icon on the results field in the Calculator Plus application, and release the mouse button. In the result, the **Object** field will contain the full name of the element for which the check will be carried out, as its screenshot appears in the **Preview** field.

Choose an object whose property will be compared.

 **Drag the target to point to the object**  
 Press the left mouse button over the target icon and drag the icon to the desired window or control. [Show me how it works.](#)


 **Point and fix (Shift+Ctrl+A)**  
 Press this button and then move the mouse pointer to the desired window, control or any other object on the screen. Press Shift+Ctrl+A to select the object. [Show me how it works.](#)

Object:

Preview:  


- Click on the **Next** button.
- On the **Select a object property to compare** panel, select the property **wText** and click on **Next**.

Select a object property to compare.

 ("CalcPlus").Window("SciCalc", "Calculator Plus", 1).Window("Edit", "", 1).wText

Properties

Basic view [View more members \(Advanced view\)](#)

[-] Standard

Enabled	True
Height	23
Name	Window("Edit", "", 1)
Parent	(Object)
Visible	True
Width	239
WndClass	Edit

[-] Extended

wSelection	
wText	7,



- In the **Specify comparison settings** panel, among the comparison settings, opt for the **Equals** option in the **Condition** drop-down, and click on the **Finish** button.

Specify comparison settings.	
Object:	Sys.Process("CalcPlus").Window("SciCalc", "Calculator Plus", 1).Window("Edit", "", 1)
Property:	wText
Condition:	Equals
<input checked="" type="checkbox"/> Case-sensitive	
Value:	7,

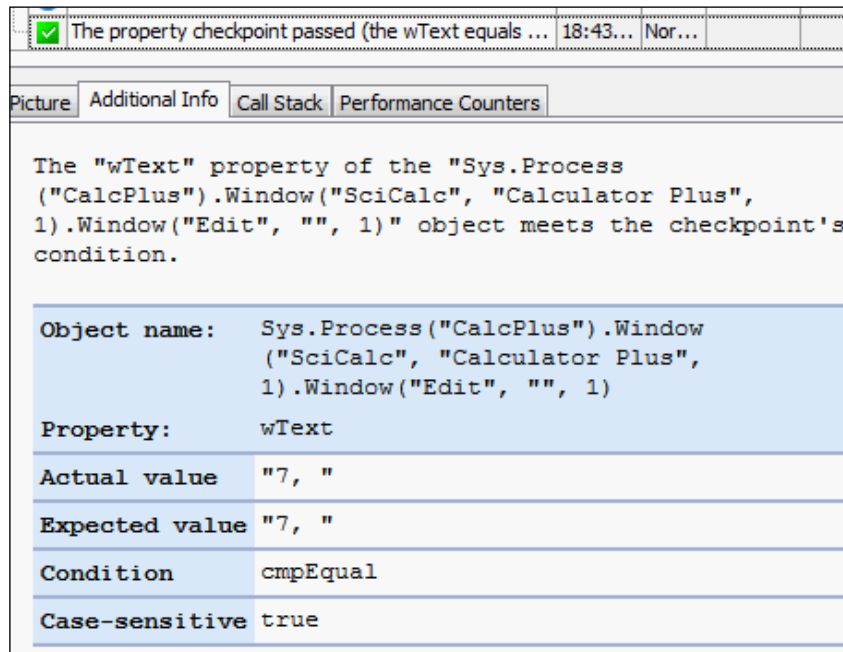
- Now, click on the **Stop** button in the **Recording** panel.
- In the result, in the TestComplete editor, we will obtain the following code:

```
function Test1()
{
    var wndSciCalc;
    wndSciCalc = Sys.Process("CalcPlus").Window("SciCalc",
        "Calculator Plus");
    wndSciCalc.Window("Button", "2").ClickButton();
    wndSciCalc.Window("Button", "+").ClickButton();
    wndSciCalc.Window("Button", "5").ClickButton();
    wndSciCalc.Window("Button", "=").ClickButton();
    var field = wndSciCalc.Window("Edit", "", 1);
    aqObject.CheckProperty(field, "wText", cmpEqual, "7, ");
}
```



We have changed the generated code a little to make the code more readable.

10. If we would launch the function, the log will have a notification, generated by the method `CheckProperty`. When the message is selected in the log, on the **Additional Info** tab, we would see the detailed description of the checkpoint.



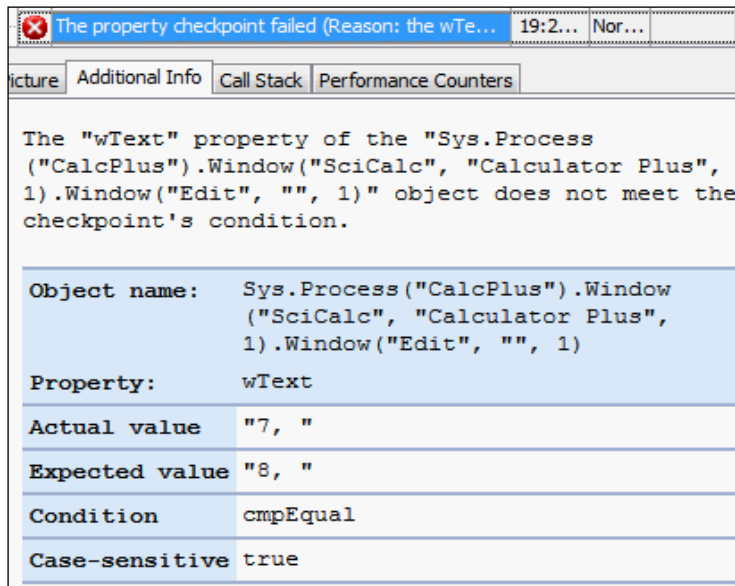
### How it works...

In the first several lines of code, we simply click on the buttons. The check itself is executed by the method `aqObject.CheckProperty`, in which function call was properly generated for us by the wizard.

The method `CheckProperty` receives four parameters:

- ▶ **Object:** This is an object, that would undergo the testing.
- ▶ **Property:** This is the property that is being verified.
- ▶ **Condition:** This is the type of the test. In our case, we have used exact comparison (the `Equals` type); however, it is also possible to use other types. For example, for the line, it is possible to check if a substring matches the line, and if it begins or ends with a specifically signified substring. For digits, one could apply fewer types of comparison. Altogether, there are about 16 types of tests.
- ▶ **Value:** This is the value itself that is expected to be obtained as a result of the check.

If the test fails, the log will contain substantial information on the possible reasons for the errors.



### There's more...

Apart from the method `aqObject.CheckProperty` there is a similar function `aqObject.CompareProperty`. The difference is in the accepted parameters: the `CompareProperty` method takes property as a parameter and it's not the object or the property as in case of the `CheckProperty` method. Moreover, the `CompareProperty` method has another parameter, **MessageType**, which allows it to assign, which type of message will be generated in the event of a failure (an error, a warning, or an ordinary notification).

## Creating object checkpoints

The purpose of testing is all about executing different types of verifications, and at times we need to check both—one property of the controls elements (for example, text in the text input field) and several other properties at a time (for example, text, availability, and visibility). To this end, there exists the object checkpoint method. It affords the checking of both several properties of a specific controls element, and several properties of its children elements.

In this recipe, we will create a object checkpoint for the Calculator Plus window along with all of its child elements (buttons and a text field).

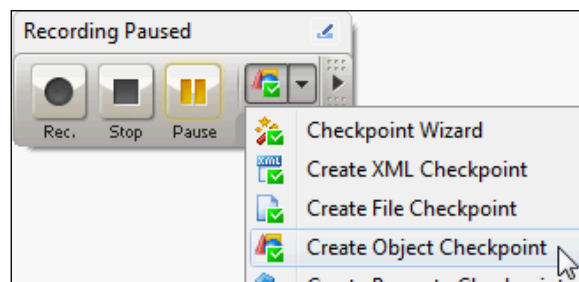
## Getting ready

Add the **Stores** element to the project by right-clicking on the name of the project, then **Add | New Item...**, and select the **Stores** element. It will be necessary for us to store the data on the objects. Then launch the Calculator Plus application.

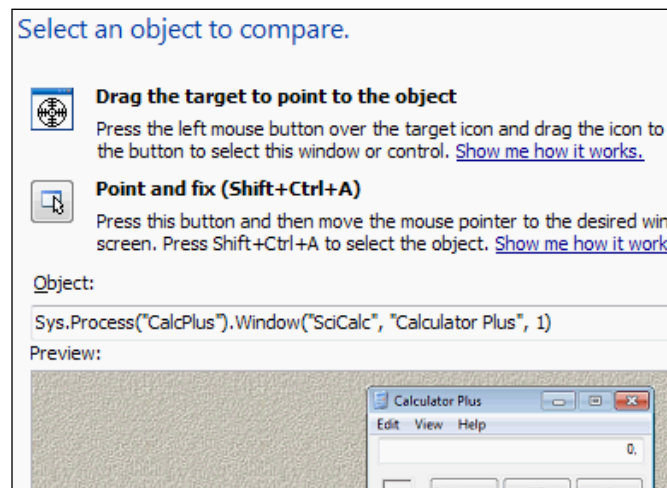
## How to do it...

The following are the steps for creating the object checkpoints:

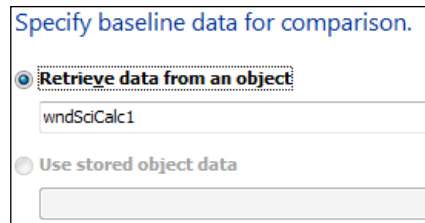
1. Begin recording the script (go to **Test | Record | Record Script** menu).
2. On the **Recording** panel, opt for the **Create Object Checkpoint** element from the **Create New Property Checkpoint** drop-down menu.



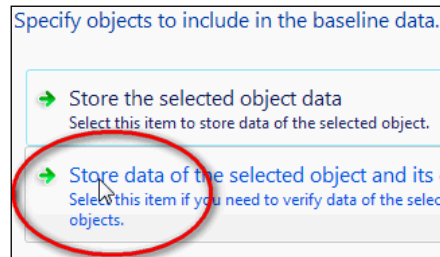
3. In the opened **Create Object Checkpoint** window, drag-and-drop the target icon onto the heading of the Calculator Plus window. In the result, the **Object** field will contain the full name of the object, and in the **Preview** field there will be the screenshot of the window.



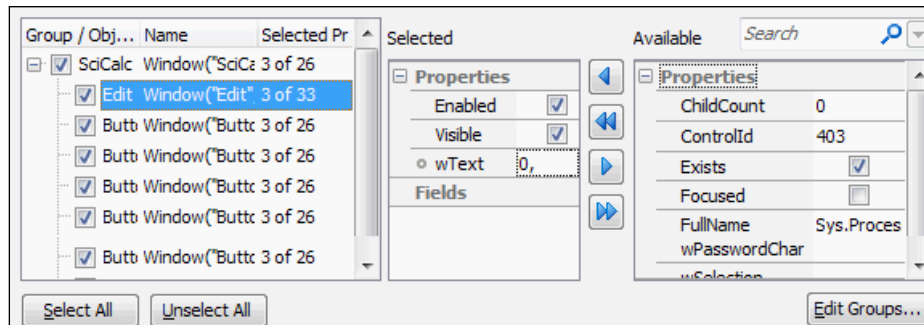
- Click on the **Next** button.
- Leave the **Retrieve data from an object** option selected by default and click on the **Next** button.



- In the next step, we can select whether we should check the information from the child windows. As we would like to check all the children objects inclusively (buttons and text fields), we select the second option, **Store data of the selected object and its children**.



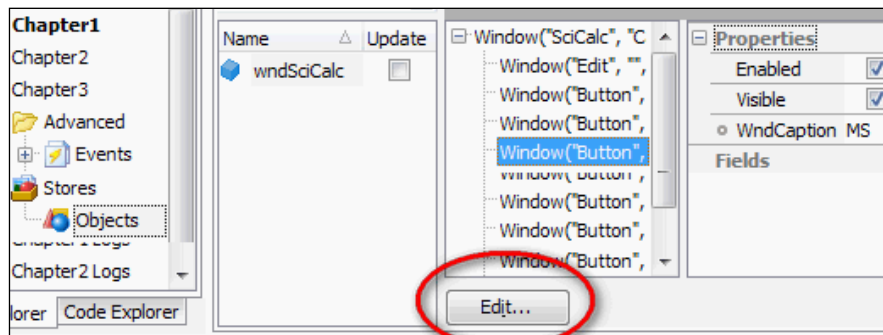
- In the result, we will have a tree view of all the objects for which the check will be performed. Let's disable all the unnecessary elements (in our case these are all the objects of the type *Static*) and click on the **Properties...** button.



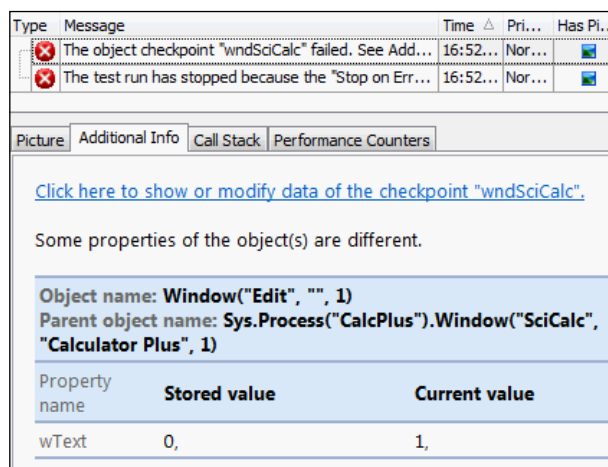
- Now, with the help of the left and right arrow buttons, for each of the elements we can add or remove the properties that are necessary to check. For the text field, we will need this properties: **Enabled**, **Visible**, and **wText**, and for the buttons: **Enabled**, **Visible**, and **Caption**, accordingly.

9. Add all the necessary properties and remove the unnecessary, after which click on the **OK** button.
10. Click on the **Finish** button.
11. Stop the recording by clicking on the **Stop** button in the **Recording** panel. In the result, we will have created a script with just a single line:
 

```
Objects.wndSciCalc.Check(Sys.Process("CalcPlus").Window("SciCalc", "Calculator Plus"));
```
12. In the **Stores** element of the project, a new **Objects** element will be created. When clicking on the element, we will see a list with just a single **wndSciCalc** element which we have just created.



13. By clicking on the **Edit...** button, we can edit the tested properties and their values.
14. If now, one wants to launch the generated function, TestComplete would compare all the selected properties of all the controls elements that are stored in the **wndSciCalc** object, with all those that are really available in the application. In case of any mismatch, an error will be generated.



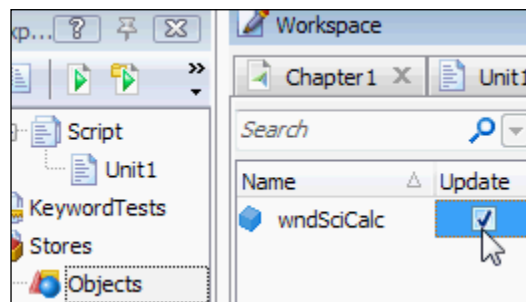
## How it works...

An object checkpoint facilitates verification of several objects' properties (or even all of them), including its child objects. The object to be verified can be either added during creation of the checkpoint (as we have done in the preceding steps), or selected from the existing objects from the **Stores** project item. In this case, the object should be added to the **Stores** element before starting to create the checkpoint. If later, the expected values of the properties change, we will not have to create the checkpoint again, but only edit the corresponding object in the **Stores** element.


## There's more...

It might happen that the object checkpoints are put to use in the project quite frequently, and sometimes, due to changes in the tested application, one has to trace back to make lots of changes to the objects. If you have encountered such a problem, and wish to work your way around making a great deal of manual changes, it is possible to launch scripts in the update mode. To do that, follow these steps:

1. Open the settings for TestComplete (**Tools | Options...**), opt for **Stores** in the **Engines** section, and turn on the **Update objects** option. Having done so, for each element in the **Objects** list that should be customized, set the **Update** option.



2. Launch the scripts, TestComplete will update all the values of the properties for the selected elements.

 Do not forget to turn off the **Update** option for all the objects after making all the necessary changes, otherwise, at each launch, updating will take place rather than checking!

## See also

- ▶ The *Creating property checkpoints* recipe

## Using global variables

Usually, to pass data between functions, parameters are used. However, this isn't always possible. In such instances, global variables come in mighty handy. These go by the names of project variables and project suite variables in TestComplete.

### How to do it...

In order to use a global variable, we need to perform the following steps:

1. Right-click on the name of the project and select the **Edit | Variables** menu item. The panel with two types of variable will open up: **Persistent Variables** and **Temporary Variables**.
2. Right-click on the empty area of **Persistent Variables** and opt for the **New Item** menu. In the result there will appear a new variable by the name of **Var1**, and the type of **String** with other empty fields.
3. Enter the default string value in the **Default Value** field, and the local string value in the **Local Value** field.

Persistent Variables				
Name	Type	Default Value	Local Value	Description
Var1	String	default string	local string	






4. Open any module and create therein the following three functions:

```
function testVariables()
{
    testExistingVars();
    testNewVars();
}
function testExistingVars()
{
    Log.Message(Project.Variables.Var1);
    Log.Message(Project.Variables.GetVariableDefaultValue("Var1"));
    Project.Variables.Var1 = "new string";
    Log.Message(Project.Variables.Var1);
}
function testNewVars()
{
```



```
Project.Variables.AddVariable("Var2", "Integer");
Project.Variables.Var2 = 123;
var variable1 = Project.Variables.Var1;
var variable2 = Project.Variables.Var2;
Log.Message(variable1);
Log.Message(variable2);
}
```

5. If the `testVariables` function is called, we will obtain the following result:

Type	Message
	local string
	default string
	new string
	new string
	123

### How it works...

In TestComplete there are two types of variables: persistent and temporary. Values of the persistent variable are saved even if the TestComplete is closed and re-opened anew. The values of the temporary variables exist only at the point of working of the scripts. On the next script run their values will be set to default.

Persistent variables can be of four types: `Integer`, `Boolean`, `String`, and `Double`. For temporary variables two additional types are available: `Table` and `Object`.

First, we have created the persistent variable `Var1` with the help of the TestComplete editor, assigning it with the default value and the current (local) value. The default value is needed in case several people are working with the same project, the ad hoc value in view will be the same across all the computers.

The function `testVariables` is needed only for the purpose of demonstrating of the fact that values of the variable may differ in one function, and then correctly show up in another.

In the `testExistingVars` function, we are working with the created variable `Var1`, first outputting its value and the default value to the log, and then changing its value and outputting the same to the log again.

In the `testNewVars` function, we consider several possibilities of handling the variable at once:

- ▶ Creating a new variable dynamically in the course of script execution, and then changing its value.

- ▶ Demonstrating a possibility for counting the value of the variable, previously changed by another function.
- ▶ We show that in the course of handling variables, it is not necessary to use their full names (for example, `Project.Variables.Var1`). And for brevity's sake, just assign the value of the newly created variable to the global variable and work with the latter, in turn. Meanwhile, changes of the new variable in no way affect the global variable.

### There's more...

Apart from the project variables, there exist project suite level variables. Working with them is no different than working with the variables of the project, with the only reservation for accessing. To access project suite level variables from scripts, one should use the code `ProjectSuite.Variables` instead of `Project.Variables`.

Similarly, to work with these variables in TestComplete itself, it is necessary to select the **Edit | Variables** menu from the **Project Suite** context menu, and not from the project. The advantage of using the variables of this type lies in the possibility of handling one and the same variable from different projects (even those using different programming languages!), and so passing data in-between.



Using global variables is considered a bad style in programming, so use it only when there is no other way to achieve the result you need.

## Testing multilingual applications

Many applications are designed for use in different countries, with different languages. Such applications usually allow switching between application languages, so that all the writings, headings, and other interface elements are displayed in the local language. More often than not, the functionality of the application as a whole is not subject to change; therefore, there is no sense in carrying out functional or regression testing for all the localized versions. Sometimes, however, the application should comply with more stringent requirements, in which case it becomes necessary to launch all the tests (or at least some of them) for all the available languages.

In this recipe, we will learn to work with multilingual application in such a way that the scripts code remains intact as we make it possible to easily switch between the languages of the tested application. Let's suppose we need to create such a possibility for a standard application of Notepad that is a part of the Windows suite.

## Getting ready

We need to perform several preparation steps before start:

1. Add the Name Mapping element to the project (right-click on the name of the project, go to **Add | New Item**, then click on **NameMapping**).
2. Make sure the automatic short names generation option has been enabled or enable it by going to **Tools | Options...**, then **Engines | NameMapping**, and check the **Map object names automatically** checkbox.
3. Launch the Notepad application.

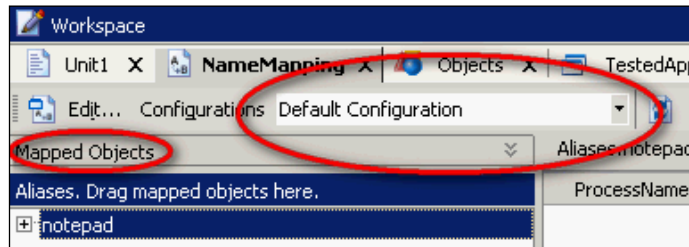
## How to do it...

In order to create a test for multilingual application, we need to perform the following actions:

1. Begin recording in TestComplete, and execute the following actions in the Notepad.
2. Input some text into the text field.
3. Then press *Ctrl + Home*.
4. Opt for the **Edit | Find...** menu item.
5. In the **Find what** field, get the word text inputted.
6. Click on the **Find Next** button.
7. And then click on the **Cancel** button.
8. Stop recording by clicking on the **Stop** button in the **Recording** panel.
9. In the result, the following script will be recorded:

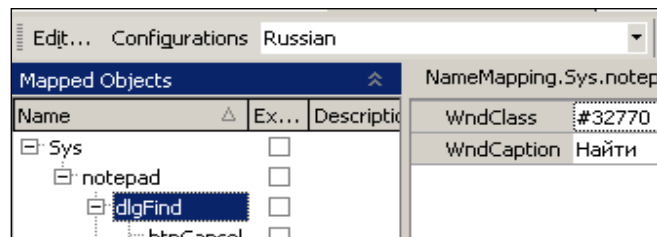
```
function testMultilanguageApp()
{
    var notepad;
    var wndNotepad;
    var dlgFind;
    var btnFindNext;
    notepad = Aliases.notepad;
    wndNotepad = notepad.wndNotepad;
    wndNotepad.Edit.Keys("Some text^[Home]");
    wndNotepad.MainMenu.Click("Edit|Find...");
    dlgFind = notepad.dlgFind;
    dlgFind.Edit.SetText("text");
    btnFindNext = dlgFind.btnFindNext;
    btnFindNext.ClickButton();
    dlgFind.btnCancel.ClickButton();
}
```

10. Click twice on the **NameMapping** project element.
11. In the right-hand side window part, click on the **Mapped Objects** heading to open it up.
12. From the **Configurations** drop-down menu on the right-hand side part of the panel, select the **Configuration Manager...** element.



You will have **Configuration Manager** window opened with one configuration (**Default Configuration**).

13. Click on the **Copy As New** button, and in the opened **Add New Configuration** window, type in a new configuration title (in our case, the name of the configuration may overlap with the language that's up for testing, for example, Russian, Danish, and so on).
14. Click on the **OK** button in the **Add New Configuration** window.
15. Close the **Configuration Manager** window.
16. Now, in the **Configurations** drop-down list on the right-hand side panel, the newly created configuration is active (if that's not so, please select it from the **Configurations** drop-down list).
17. Expand the **Sys | Notepad** elements in the **Mapped Objects** panel, and select the **dlgFind** element.
18. In the right-hand side part, change the **WndCaption** property in such a way that it tallies up with the heading of the **Find** window for the selected language.
19. Click on the button with the ellipsis to the right of the property value and in the opened **Edit the WndCaption Property Value** window, change the value of the **Value** field, and then click on the **OK** button. An example of the **Russian** configuration is shown in the following screenshot:



20. In the same way, change the properties for all the controls elements, in which the text is to be changed at the point of changing the application language (in our case, these are **btnCancel** and **btnFindNext**).
21. Now, if we launch the previously recorded script, we will see this error in the log: **The control item 'Edit' not found**. That has happened because we have changed the headings for all the controls elements that are available in the **NameMapping** element; however, the names of the menu items have been written directly in the code, and they were not possible to transpose to **NameMapping**. The simplest way to resolve this problem is to create two objects that will contain the names of the menu items in different languages:

```
var menuEng = {
    Edit: "Edit",
    EditFind: "Edit|Find..."
};
var menuRus = {
    Edit: "Правка",
    EditFind: "Правка|Найти..."
};
```

22. We will place these two objects outside our function and the function should be then modified in the following manner:

```
function testMultilanguageApp()
{
    var menu = menuRus;

    var notepad;
    var wndNotepad;
    var dlgFind;
    var btnFindNext;
    notepad = Aliases.notepad;
    wndNotepad = notepad.wndNotepad;
    wndNotepad.Edit.Keys("Some text^[Home]");
    wndNotepad.MainMenu.Click(menu.EditFind);
    dlgFind = notepad.dlgFind;
    dlgFind.Edit.SetText("text");
    btnFindNext = dlgFind.btnFindNext;
    btnFindNext.ClickButton();
    dlgFind.btnCancel.ClickButton();
}
```

## How it works...

**NameMapping** can contain as many configurations as possible, each one corresponding to the languages of the tested application. Perhaps the most difficult aspect of handling such applications is manual renaming of all the properties of all the elements used in **NameMapping**; however, this should be done just once. In order to launch the scripts for another application language, it is sufficient to change the current configuration on the **NameMapping** panel, and have the possibility to quickly change the language that is being used for the menu (both for the main and all other contextual menus of the application that are evoked by the right-click). In our case, we have decided to form up the elements of the main menu as objects.

A more successful solution for information storage would be the usage of **data-driven** approach. This would allow inputting all the data on the menu into separate files with tables (for example, each language would match individual Excel files). Implementation of such an approach, however, would be overkill for the given recipe.

## There's more...

Changing configurations for the **NameMapping** element is possible not only from the TestComplete window, rather directly from the script code. To this end, it is necessary to set the required value for the **CurrentConfigurationName** property of the `NameMapping` object. For example, switching the Russian configuration on would have the following effect:

```
NameMapping.CurrentConfigurationName = "Russian";
```

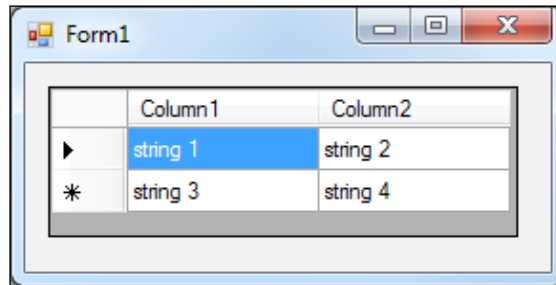
## See also

- ▶ You can read more about the data-driven approach in the *Chapter 9, Data-driven Testing*.
- ▶ If you want to learn more about how `NameMapping` and `Aliases` are working, refer to the article at <http://support.smartbear.com/viewarticle/27370/>.

## Working with nonstandard controls

Sometimes, the possibilities extended by TestComplete are not sufficient to properly handle the controls elements. If that's the case, we need to extend TestComplete possibilities by using private properties and methods of objects that are within TestComplete's access. The brightest example of the situation at hand lies in working with the grids. Grids are quite complex controls elements, changeable and customizable, depending on the situation.

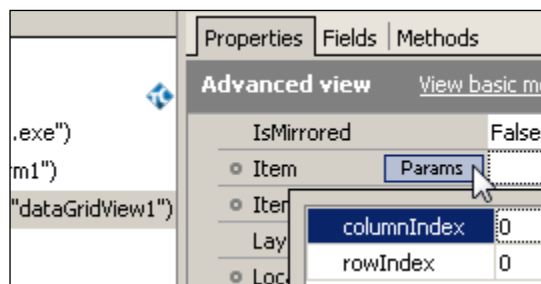
In this recipe, we will use a simple example of the Microsoft DataGridView control to see how standard control properties can be used for solving nonstandard situations. Our task consists of placing a screenshot to the log of a single cell of the grid (by pre-assigned column and row number). TestComplete allows making a screenshot only of the whole grid, but combining native properties of the control with the TestComplete possibilities, we will be able to come by the targeted result. As a tested application, we will use a simple .NET application, which has only one grid with two rows and two columns.



## How to do it...

In order to create a code which gets a screenshot of a specific cell, we need to perform the following steps:

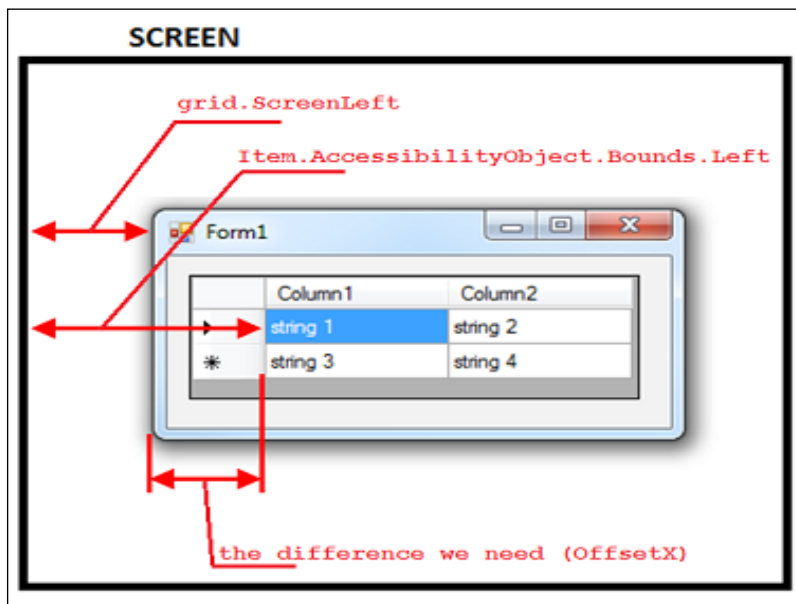
1. First of all, we will need to clarify which means are at our disposal. TestComplete has the `Picture` method, which returns a screenshot of the controls elements, for which it has been called. This won't do us any good, however, as this method has optional parameters (`ClientX`, `ClientY`, `OffsetX`, and `OffsetY`), which allow making a screenshot of its specific region impossible.
2. Now we will have to ascertain if we could get a hold on the coordinates and the size of the specific cell in the grid. By carefully looking into the properties of the grid with the help of the **Object Browser** tab, we will locate the `Item` property that takes two parameters—number of the row and the number of the column—and returns the object that corresponds with these coordinates.



- In turn, this object has a property `AccessibilityObject`, while it has another property `Bounds`, which just fills the bill with the necessary information. Thus, the full path to the left coordinate of the grid will be `WinFormsObject("dataGridView1").AccessibilityObject.Bounds.Left`.

Here, however, we encounter a difficulty: the given properties of `Bounds` are stored in relation to the screen (for example, the property `Bounds.Left` contains the X-coordinate relative to the left-border of the screen), and not relative to the grid. This means, we will have to transform the absolute screen coordinates that are returned by the object of `Bounds`, into relative ones.

- Again, having perused the properties of the grid, we will get around to its properties of `ScreenLeft` and `ScreenTop`, which also spell out absolute positioning in relation to the screen. This way, subtracting the coordinate `grid.ScreenLeft` from the cell coordinate `Item.AccessibilityObject.Bounds.Left`, we will get the distance measured in pixels from the left border of the grid to the cell. This is exactly what we need to pass to the `Picture` method.



- In exactly the same manner we will calculate the vertical coordinates (to this end, we will use the `grid.ScreenTop` and `Item.AccessibilityObject.Bounds.Top` properties).
- While dealing with the two more parameters of the `Picture` method (`OffsetX` and `OffsetY`)—they simply correspond to the width and height of the cell, and we can obtain them directly from the `Bounds` property.



7. In the result, the function that returns the screenshot of the cell will be constructed programmatically to the following effect:

```
function getGridCellPicture(grid, row, col)
{
    var cellBounds = grid.Item(col,
        row).AccessibilityObject.Bounds;
    var xOffset = cellBounds.Left - grid.ScreenLeft;
    var yOffset = cellBounds.Top - grid.ScreenTop;
    return grid.Picture(xOffset, yOffset, cellBounds.Width,
        cellBounds.Height);
}
```

8. Looking up the example of the function at work is possible by writing up a simple function.

```
function testGrid()
{
    var wnd =
        Sys.Process("TestGrid").WinFormsObject("Form1");
    var grid = wnd.WinFormsObject("dataGridView1");
    wnd.Activate();
    Log.Picture(getGridCellPicture(grid, 0, 1));
}
```

9. In the result of the preceding function, the screenshot of the cell will be found in the log with the coordinates (0;1) (that is, from the first row of the first column, as numbering starts with zero).

## How it works...

Despite the seeming simplicity of the solution of the task, writing a similar universal function may take up from several hours to several days on end. The whole matter is that locating the necessary property and understanding how it works is quite a difficult task, especially so when it has never been done before. Another factor that influences speed of the solution of a similar task is the internal complexity of the controls element that you are handling.

Moreover, the solution that suits a specific instance may be a misfit for another. Then a programmer has to tentatively try various solutions. Unfortunately, in such cases there is no other way around. We come up with a solution and try to implement it by looking for necessary properties and methods. One should be looking with the help of **Object Browser** or in the documentation on the controls element that is being worked over. Often, the selected approach turns out to be improper, and we have to keep looking for newer ways towards the solution.

Sometimes, it is useful to request a consultation from programmers that are coding the tested application, as they are working with these elements on a day-to-day basis and may give you a hint off the top of their heads. Nonetheless, it should be known that programmers practically never go about such tasks, for example, getting coordinates (just the thing we had to accomplish); and the best-case scenario would be combining efforts of programmers and testers.

### There's more...

The given example is quite simple, however, in real projects one should account for many additional factors. For example, what would happen if the passed number of the row is greater than the overall number of the rows? What would happen if the cell or its part is found outside the ambits of the screen or beyond the border of the controls element? What would happen if the cell is hidden with the settings of a filter in the grid? All of these points are quite realistic, which may lead to emergence of errors, and this is why such functions should be written really carefully, thought through, and reproduced in as many scenarios as possible.

### See also

- ▶ *The Mapping custom control classes to standard ones* recipe in *Chapter 5, Accessing Windows, Controls, and Properties*
- ▶ *The Using text recognition to access text from nonstandard controls* recipe in *Chapter 5, Accessing Windows, Controls, and Properties*
- ▶ *The Using Optical Character Recognition (OCR)* recipe in *Chapter 5, Accessing Windows, Controls, and Properties*

## Organizing script code in the project

When working with the real project, it is necessary to somehow organize code repository to be able to easily locate the code. This can be achieved through placing functions and variables that are bound to a specific functionality, into separate units. This approach is called **functional decomposition**.

In this recipe, we will take on an example of such decomposition for testing the Calculator Plus.

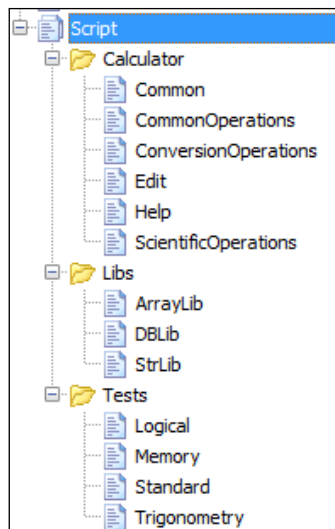
## How to do it...

1. First of all, we need to find out which layers are going to break down the code of our project. Usually, there are three such levels:
  - **The level of the libraries:** In this level, we create functions and classes that may be used in any project. For example, functions operating with strings, arrays, and databases (unbound to any specific database) and functions created for simplification of the data-driven approach, for working with the e-mails and another code that is liable to come in mighty handy not only in the given project, but in any project (both existing and prospective ones).  
On this level, the units may apply functions from other units, only being on the same level (that is, on the level of the libraries).
  - **The level of the application's common code layer:** Here we will go about creating functions to work with the application (or several applications) of the given project. For example, launching and terminating the application, login functions, and various checkers that are specific for the project.  
Functions of this level can be used by other functions of the same level and the functions from the libraries (as this is the original intent of them having been created). Functions of this level in their turn, will be used on the next level of the tests.
  - **The level of the tests:** In this level, we store only those tests, for example, scripts, which are meant to execute specific actions within the application. Functions of this level can apply other functions of the level of the libraries, and functions of the level of application code.
2. These three layers in TestComplete are easier to implement with the help of the folders that are created in the **Scripts** element. On each level, we will have several files, and that's why it would be logical to place them in separate folders.  
Let's consider each level individually.
3. Everything is quite simple with the level of the libraries. Here, for each of the library types, we have a separate unit; moreover, the name of each unit will correspond to the original intent of the inherent functions. For example, `StrLib` is meant for working with the strings, `DBLib` in its turn is meant for working with databases, and so on. If such a level of abstraction is not sufficient (for example, when we have different functions for working with databases Oracle and MS SQL), one can create another folder within the libraries folder, and place therein several individual units, each having a set of functions for a singular database.
4. Level of the application code is more complex and depends much on the complexity of the tested application itself. Here, we need to break the code down into logical and functional applicability. For example, in order to test Calculator Plus, we may have the following units:

- ❑ **Common:** This unit can be used for functions of the generic intent: for opening or closing the application, clearing the results field, and so on.
- ❑ **CommonOperations:** This unit can be used for operations that are available from any modulus operandi (summing up, subtraction, and work with the memory).
- ❑ **ScientificOperations:** This unit can be used for operations that are available only in the scientific mode (logarithms, exponentiation, working with the corners, and different numerical systems).
- ❑ **ConversionOperations:** This unit can be used to work with the functions that are available in the conversion mode only.
- ❑ **Edit:** This unit will comprise functions that correspond the **Edit** menu (copying and pasting).
- ❑ **Help:** This unit can be used to work with the **Help** menu (evoking the **Help** and **About** windows).

This structure can be changed as the tester sees fit, with the only reservation about the code being logically layered in the project.

5. In the level of tests, a certain clear-cut structure should also be pursued to steer clear of confusion in the existing tests. For example, it is possible to store tests by organizing them according to the same principles that are applied to structure the application code. If you have some manual tests that you are trying to automate, stored according to an existing logic (for example, in the bug tracking system), you can duplicate the same structure into the TestComplete project. There are no stringent rules here, it is only important to avoid chaos.
6. An example is depicted on the following screenshot concerning the way a project may appear in order to test the Calculator Plus application.



## How it works...

At first glance, it may appear that for such a small application as **Calculator** or Notepad, there is no sense in creating such a structure, however, the truth is just the opposite. The structured code is much easier to maintain and build up; and it's much easier to get your bearings straight in efficiently thought-through project. A small application with time may grow into a larger one, medium size application and then into an enormous one: if the rules are broken from the beginning, with time it will only be more difficult to recollect the whereabouts of things and the underlying reasons. The method of functional decomposition also allows creation of readable tests. Such tests consist of functions calls of the lower-level and grasping them is much easier than just written code, as the names of the called functions are usually self-explanatory. Such test is of easy understanding even for a person who is not well cognizant in programming.

## There's more...

If you have several different projects in your company that are applying TestComplete but that do not intersect with each other, it does make sense to organize the libraries into a separate project and store it separately. This, however, requires discretion at the point of making changes, since these changes will affect everything.

Also note that, changes in the application under test will affect the script code as well.

## See also

- ▶ *The Creating framework using the OOP approach recipe*

## Handling exceptions

In case of emergence of any exceptional situation during script execution, TestComplete will stop the execution and display the error message. For example, when attempting to open a non-existing file, we will get an error message **Unable to open the file**.

To resolve this problem and continue script execution, we can apply a standard `try...catch...finally` construct.



This recipe contains information which is only applied for JScript programming language!

## How to do it...

For example, let's suppose we need to read the contents of a file into a variable, if the file exists, after which the file should be deleted and a new eponymous empty file should be created.

1. First of all, let's write up the code that is responsible for reading the contents of the original file and then deletes the same:

```
var fileContent = "";
var fileName = "c:\\non-existing-file.txt";
try
{
    var f = aqFile.OpenTextFile(fileName, aqFile.faRead,
    aqFile.ctUTF8);
    fileContent = f.ReadAll();
    f.Close();
    aqFile.Delete(fileName);
}
```

This code has been placed into the `try` block at once, because it is fraught with an occurrence of an exceptional situation if the file does not exist. In this case, we need to simply ignore the exception by logging the occurrence to the log (in case, we have to analyze the flow of script execution later).

2. Now, let's write up the contingency code for processing the exceptional situation.

```
catch(ex)
{
    switch(ex.number)
    {
        case -2147467259:
            Log.Message("File does not exist");
            break;
        default:
            throw ex;
    }
}
```



Prior to ignoring the exception, we check its code (`ex.number`) and proceed with ignoring the exception only in the case that its number matches with the code of the *missing file* error (that is, `-2147467259`). If, at this point, another exception takes place (for example, the file is existing, but it has been blocked by another application), we will generate the exception anew with the help of the `throw` instruction, since we do not know how to further treat the situation.

3. And in the end, in the final block `finally`, we will execute actions that it is necessary to execute regardless of whether the exception has taken place or not:

```
finally
{
    aqFile.Create(fileName);
}
```

4. Here, we create a new file without second thoughts about the file with the same name that may already exist: in first case we delete the file ourselves, and in the second case the file was not in existence to begin with.

If we were to merge all the written code into a single function and launch it, the function would work successfully regardless of the fact whether the `c:\non-existingfile.txt` file exists or not.

### How it works...

The `try...catch...finally` block is meant to process various nonstandard situations when we have a little knowledge of how the code will end up working:

- ▶ In the `try` block, the code with potential errors is placed.
- ▶ In the `catch` block, we handle these errors and continue to carry out all the necessary actions (for example, in our case, it is enough to enter the information on the arisen exception to the log). If we need to ignore all the errors, this block could be simply left empty, however, this is considered bad style in programming.
- ▶ In the `finally` block, we are busy carrying out the actions which should be performed regardless of the emergent situation. This block is optional; however, there may be some situations where it is indispensable. For example, if we have opened a file and get some actions done over it, it has to be closed despite any possible errors that emerged, otherwise, this may tamper with the filesystem integrity.

### There's more...

`TestComplete` does not signify the number of the emergent errors upon exceptional situation, which means that in order to find out the code of the error we are interested in, we first resort to the `catch` block to the following effect:

```
catch (ex)
{
    Log.Message(ex.number);
}
```

After this, the function has been called to make sure the file is missing on the disk. In the result, in the log we had the error's number, which was then used in the `switch` construct.

If, further on, we are up against other situations, which we would like to ignore or process in a different manner, we would simply add another block `case` and write all the necessary actions into it.

## See also

- ▶ *The Handling exceptions from a different unit recipe*

## Handling exceptions from a different unit

The engine of the Jscript language, used in TestComplete, does not support working with several units. Such a possibility is provided by the TestComplete itself. One of the side effects consists in the following situation: if we are in one unit inside the `try...catch` block calling the function from another unit, and this second function raises an exception, it will not be caught by the `try` block. In the result, TestComplete will stop script execution.

The simplest way to solve this challenge is creation of the `try...catch` block in the called function and using the returned value of the function for analysis in the function-call. This solution, however, may misfire in several situations, this is why in the given recipe we will consider one of the methods to intercept exceptions in the functions of another unit.

## Getting ready

Before we start, we need to perform the following preparation steps:

1. Create two new units **UnitA** and **UnitB** (right-click on the **Script** element, and go to **Add | New Item**).
2. Place the following function into **UnitA**:

```
function calledFunction()
{
    Log.Message("Called function from UnitA");
    throw "exception from UnitA";
}
```

We will call this function from another unit.

3. Into the **UnitB** unit, place the following code:

```
function callerFunction()
{
```



```
Log.Message("Message from caller function");
try
{
    UnitA.calledFunction();
}
catch(ex)
{
    Log.Message("Exception caught!", ex);
}
}
```

In this function we are calling the `calledFunction` function from the **UnitA**, meanwhile trying to intercept all the arisen exceptions. If this function is launched, we would see that the exception is not intercepted, rather the following error message is displayed on the screen: **Exception thrown and not caught**.

Namely this problem we are supposed to resolve right now.

## How to do it...

In order to catch the exception from a different unit, we need to perform the following steps:

1. Create a new `call` function and place it into any of the two existing units:

```
function call(functionName)
{
    var fnArray = functionName.toString().split("\n");
    fnArray[0] = fnArray[1] = fnArray[fnArray.length-1] = "";
    return fnArray.join("\n");
}
```

2. In the `callerFunction` function replace the direct call of the function (`UnitA.calledFunction()`) with the following programming construct:

```
var str = call(UnitA.calledFunction);
eval(str);
```

3. Launch the `callerFunction` function. In the result, the error message will not appear, and in the log, we will have the following message written down: **Exception caught!**. If you select this message in the log, on the **Additional information** panel, we will see the text of the caught exception that is generated by the `calledFunction` function: **exception from UnitA**.

## How it works...

Since exceptions from other units are not intercepted, our task consists of launching the code we are interested in from the current unit via the instrumental function call of the additional `call` function. In JScript language everything is an object, including a function. This is why we can pass the function as a parameter. Then, with the help of the `toString` method, we obtain the code of the very function as an ordinary text string. With the help of the `split` method, we transform this string into an array so that in the future we could remove all the unnecessary strings from it (the first two lines are the declaration of the function and the opening curved bracket, as well as the closing curved bracket). These lines are, for the sake of simplicity, to be replaced with empty lines.

In the result, in the array, we have only the body of the function left. The array will be transformed back into the string with help of the `join` method, and it will return the resulting string of code back to the `callerFunction` function). Further, with the help of the built-in `eval` function, we execute the code that was returned as a string with the `call` function. Since, this code is in fact executed in a single unit **UnitB**, we have a possibility to intercept the exception.

## There's more...

Despite the fact that examined approach allows resolving a fairly complex problem, it is easy to see its shortcomings:

- ▶ The given example allows launching only a function without parameters. Writing of the same function with parameters would significantly complicate the code of the `call` function.
- ▶ The function implies that the code conforms to a specific style (the first line is the heading of the function, and the next line is the opening bracket). To better transform the function into the executable code, one would have to write up a smarter function.
- ▶ In the result of adding the new function, we have in fact two lines of code instead of one line. Although, this could be simply resolved by joining the two lines of code into a single one:

```
eval(call(UnitA.calledFunction));
```

## Creating framework using the OOP approach

**Object-oriented Programming (OOP)** is widely used not only in program development, in automated testing as well. With its help, tests become more readable and simple in maintenance and follow up.

In the given recipe, we will consider an example of creating a simple object-oriented framework for testing the Notepad application and writing up a simple test applying the OOP approach. The task of the test is launching the Notepad application, open the existing file, check if the file has opened, and then close the Notepad application.

### Getting ready

Perform the following steps before starting:

1. Add the **TestedApps** element to your project (right-click on the project name, and select **Add | New Item**).
2. Add the standard Windows Notepad application to the **TestedApps** element (right-click on the **TestedApps** element, then select **Add | New Item**, and enter this path `C:\Windows\notepad.exe`).
3. Create a file with the name of `C:\testfile.txt`, in which only one string should be written: `test string`.

### How to do it...

In order to create a simple OOP test, we need to perform the following steps:

1. First of all, it is necessary to think through the structure of the class for our application. We will need four methods for working with the Notepad application:
  - `start`: This method is used to launch the Notepad application
  - `close`: This method is used to close the Notepad application
  - `openFile`: This method is used to open the file
  - `checkContent`: This method is used to check the text in the Notepad application

We will also need two methods to access the windows of the application:

- `wMain`: This method is used to access the main **Notepad** window
- `wOpen`: This method is used to access the **Open** window

2. The template of our class will look as follows:

```
function Notepad()
{
    this.wMain = function() {}
    this.wOpen = function() {}

    this.start = function() {}
    this.close = function() {}
    this.openFile = function(fileName) {}
    this.checkContent = function(content) {}
}
```

3. Now, we will consider each method separately. We have specifically implemented them to the simplest effect ever, so that there's no need to dwell on the particulars of the realization of the methods by and large.
4. The `wMain` and `wOpen` methods simply return the main window of the Notepad application and the dialog window **Open**.

```
this.wMain = function()
{
    return Sys.Process("notepad").Window("Notepad", "*");
}
this.wOpen = function()
{
    return Sys.Process("notepad").Window("#32770", "Open");
}
```

5. The `start` and `close` methods are also quite simple ones.

```
this.start = function()
{
    TestedApps.notepad.Run();
}
this.close = function()
{
    TestedApps.notepad.Terminate();
}
```

6. The `openFile` method gets more complicated and includes several actions: menu selection, file name input, and clicking on the **Open** button.

```
this.openFile = function(fileName)
{
    this.wMain().Activate();
    this.wMain().MainMenu.Click("File|Open...");
    var fileEdit = this.wOpen().Window("ComboBoxEx32").
Window("ComboBox").Window("Edit");
    fileEdit.Keys(fileName)
    this.wOpen().Window("Button", "*Open").Click();
}
```

7. And, finally, the `checkContent` method is also very simple, as it compares real value in the text field with the expected one.

```
this.checkContent = function(content)
{
    var editArea = this.wMain().Window("Edit");
    aqObject.CompareProperty(editArea.wText, cmpEqual,
content);
}
```

8. Now, we have a simple class to work with Notepad, and we can get down to writing up the test. The test will consist of just four steps: open the Notepad application, open a file within Notepad by a specified filename, and check the contents of the opened file, after which Notepad will be closed. The test will look as follows:

```
function TestNotepad()
{
    var expectedContent = "test string";
    var wNotepad = new Notepad();
    wNotepad.start();
    wNotepad.openFile("c:\\testfile.txt")
    wNotepad.checkContent(expectedContent);
    wNotepad.close();
}
```

## How it works...

Peculiarity of the object-oriented approach to test creation consists of encapsulated realization inside the methods so that the tests themselves would be more simplistically written, maintained, and followed up.

---

Pay attention to all the available methods in our class. If we had recorded the same actions, the code would consist of tens of lines and would be more difficult to understand. In case of the OOP approach, the test consists of just four lines of code, each being understandable effortlessly, since the names of the methods are self-explanatory.

Ideally, working with the tested application in the test should consist of calls of various methods of the class only. This is not always possible, but this should be strived for. Of course, this method has its drawbacks. For example, TestComplete does not support object-oriented possibilities of Jscript language, that's why you won't be able to see the names of the methods in the drop-down auto-filled list, and won't be able to move to the necessary method by keeping the *Ctrl* key pressed and clicking on the name of the method. However, these drawbacks are well made up for by convenience of handling the tests that have been written in the preceding mentioned manner.

Another gap is that the writing effort increases in case of growing number of GUI controls to describe. But in return we're gaining an advantage in maintainability.

### There's more...

JScript and VBScript languages support object-oriented programming. The DelphiScript language is short of such possibilities. This is why, if you are set upon usage of OOP approach together with the DelphiScript language, please use the **ODT (Object-Driven Testing)** object extended by TestComplete.

### See also

- ▶ The *Organizing script code in the project* recipe



# 4

## Running Tests

In this chapter we will cover the following recipes:

- ▶ Running a single function
- ▶ Verifying test accuracy
- ▶ Creating a test plan for regular runs
- ▶ Running tests from the command line
- ▶ Passing additional parameters to test from the command line
- ▶ Organizing test plan runs
- ▶ Scheduling automatic runs at nighttime
- ▶ Running tests via Remote Desktop
- ▶ Changing playback options
- ▶ Increasing run speed
- ▶ Disabling a screensaver when running scripts
- ▶ Sending messages to Indicator
- ▶ Showing a message window during a script run

### Introduction

All the tests we create should be run as often as possible. It doesn't only allow to test the application, but also to stabilize our test scripts. By constantly running and improving the tests, we make sure that they only fail in case there are problems with the tested application, not the scripts themselves.

In this chapter, we will consider different ways of running test scripts and organizing script runs.



## Running a single function

During writing code or debugging a test, we, from time to time, would launch it in order to make sure that we are doing everything right and the test is working properly, according to our original intent.

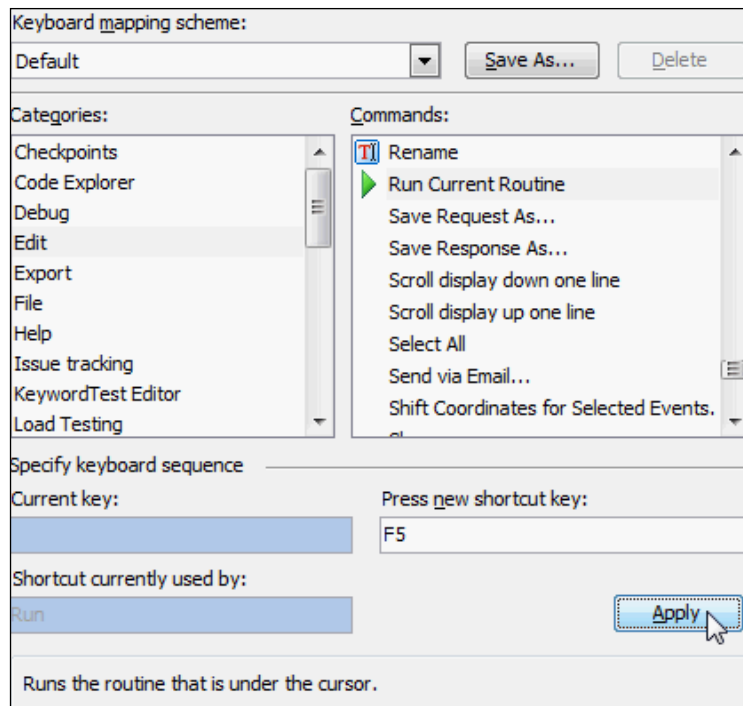
### Getting ready

Create a function, which executes a simple action (for example, enters a message into the log with the help of the `Log.Message` method).

### How to do it...

In order to run a function, we will perform the following steps:

1. Right-click on any place in the function that you would like to launch.
2. Select the **Run Current Routine** option. In the result, the function will be launched. By default, for this option there is no predefined keys combination, however, we can easily create it if we wish to launch the function at a single key stroke.
3. Now, select the **Tools | Customize Keyboard...** option.
4. In the **Categories** list, select the **Edit** option.
5. In the **Commands** listing, go for the **Run Current Routine** element.
6. Click on the **Press new shortcut key** field and press the key or combination of keys which you would like to use to launch the current function. If the selected combination is already in use and thus reserved for another action, the name of the action will be displayed in the **Shortcut currently used by** field.
7. Click on the **Apply** button.



8. If in the **Keyboard mapping scheme** field, the **Default** scheme has been selected, TestComplete will suggest creating a new one, since that scheme is a read-only one. Click on **Yes** and assign the name of the new scheme in the **Create a New Keyboard Scheme** window.

Now, we can launch the currently selected function with the use of the selected key or keys combination.

### How it works...

We can only run those functions which do not accept parameters, otherwise the **Run** option will be disabled. The default keyboard mapping scheme always remains read-only so that we can easily roll back the changes if necessary.

## Verifying test accuracy

After having written a test and done all the necessary checks, it is necessary to make sure that, at least those cross-checks which we have explicitly assigned are working properly. The simplest way to get this done is to change the expected values to the opposite ones (or different ones) and launch the test again. Let's suppose that one of the working conditions of our tests is launching them on the 64-bit system, which is the first check we are making in our tests.

### Getting ready

Write up the following function that checks the bit rate of the system:

```
function isWin64()
{
    aqObject.CompareProperty(Sys.OSInfo.Windows64bit, cmpEqual,
    true);
}
```

At the point of launching our test this check will successfully go over and in the log there will appear messages about the verified values.

### How to do it...

To verify test accuracy, we will need to perform the following steps:

1. Change the expected value to a different one. In our case, we will replace `true` with `false`:  

```
aqObject.CompareProperty(Sys.OSInfo.Windows64bit, cmpEqual,
false);
```
2. Now, we will launch our test and make sure that in the log we indeed receive the warning or an error message, as initially intended by test conditions.
3. If no error message or a warning appears in the log, this means that the check has not worked correctly (for example, a wrong property or controls element has been selected for the check); and we should make the necessary changes in the test.

## How it works...

We check all the checkpoints throughout the test in the same way, after which the test can be added to the testing plan for routine launches. This ought to be done in order to verify that the checks in the test are working properly, concordant to the original intent at creation. It is also possible to emulate other situations (for example, as the test is working, close the window manually (the one the test is checking) and see that the log has a clearly generated entry about the error). If these checks are not being done, it may turn out that the created tests never generate error messages, even though there are errors to be found in the application. We have no use for such tests.



After all the undertaken checks, correct values should be returned in the result!

## See also

Checkpoints are explained in details in the following recipes:

- ▶ The *Creating property checkpoints* recipe in *Chapter 3, Scripting*
- ▶ The *Creating object checkpoints* recipe in *Chapter 3, Scripting*

## Creating a test plan for regular runs

The written tests should be regularly launched, wherein it is preferable to have a handy tool for organizing and launching the tests. In TestComplete, there is **Test Items** project element that is meant for organizing and launching the available tests.

In this recipe, we will learn how to add Test Items to the project and customize tests parameters.

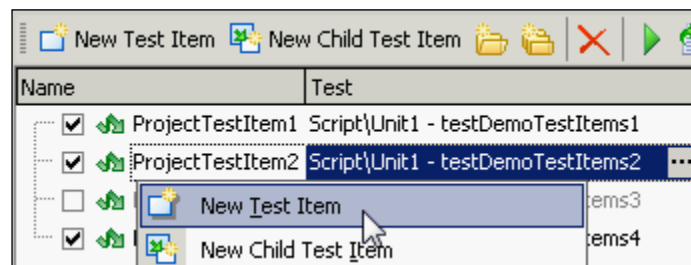
## Getting ready

Create several functions which will be treated as tests to be added to **Test Items**. They can contain any actions or even lead to no action at all.

## How to do it...

In order to create test plan, we need to perform the following steps:

1. Right-click on the project name and select the **Edit | Test Items** option. In the result, the **Test Items** panel with an empty list will be opened in the right-hand side of the **TestComplete** window.
2. Click on the **New Test Item** button on the workspace toolbar above the list. In the result, a new element with the default name as **ProjectTestItem1** will be added to the list. This is the name of our Test Item. It can be changed by clicking once on the name and by entering a new name therein.



3. Click on the button with the ellipsis in the **Test** column. The **Select test** window will open, select the **Script** element, and then the unit, in which the necessary test is to be found.
4. In the **Available Tests** column, opt for the function you would like to map to the created Test Item, and click on **OK**.
5. After this we readily have a prepared Test Item that can be launched using one of the two methods:
  - By selecting the **Run Focused Item** item from the context menu
  - By running the whole project (right-click on the name of the project and run the project).

## How it works...

Test Item is a special element of the project, which is designed for easy management of tests launches. To the left-hand side of each Test Item there is the **Enabled** checkbox, which allows at a single mouse-click to exclude the tests from being launched.

Besides, for each of the Test Items there are two more interesting customizations: **Count** and **Timeout** (they are assigned in the corresponding columns). The first one of them allows launching the test several times consecutively when needed, while the second method allows setting time in terms of minutes, upon expiry of which the test will be considered as a hung-up and stopped with a corresponding error entered in the log. Meanwhile, all the other tests will be launched as usual.

Moreover, if the tests accept some parameters, they can be assigned in the **Parameters** column. All the Test Items are independent of each other and in particular, on the functions they are bound to, this is why it is possible to create several Test Items that would launch one and the same function (for example, with different parameters or at a different time).

Working with the Test Items is possible directly from the tests, as the `Project.TestItems` object is being used to this end. For example, in the following code, we place the name of the current Test Item into the log:

```
Log.Message(Project.TestItems.Current.Name);
```

### There's more...

The only inconvenience of using Test Items is an impossibility of launching separate Test Item from the command prompt (see the recipe *Running tests from the command line*); this could be done only for a separate function, nonetheless, we are devoid of the possibility to set the timeout.

### See also

Test Items are usually used for running tests regularly from command line. These questions will be covered in the following recipes:

- ▶ The *Running tests from the command line* recipe
- ▶ The *Organizing test plan runs* recipe

## Running tests from the command line

One of the most important tasks in test automation is a possibility to launch tests from the command prompt (for example, in order to have tests run automatically at a preset time or embed launching tests to the system at application compilation). TestComplete supports a wide variety of parameters for usage of the command line prompt (for launching tests and some additional customizations). In this recipe, we will deal with two possibilities: launching a separate function and launching the test plan.

## Getting ready

Create several functions with any contents (even empty functions will do, that do not execute any actions whatsoever) and add these functions to Test Items (this we have seen in the *Creating a test plan for regular runs* recipe).

## How to do it...

In order to run tests from command line, we need to perform the following steps:

1. First, let's launch all the project tests to be executed.
  1. Launch the interpreter of the command line (for example, click on the **Start** button, write `cmd` in the search box, and press *Enter*).
  2. Write the following command in the interpreters window:

```
"C:\Program Files\SmartBear\TestComplete 9\Bin\TestComplete.exe" z:\TestCompleteCookBook\TestCompleteCookBook.pjs /run /project:Chapter4
```

Here, we have a full path to the signified executable TestComplete file.



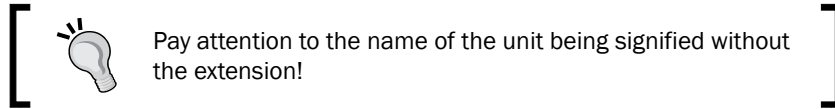
Please pay attention to the double quotation marks, they are necessary since the file path contains spaces.

2. Then goes the name to the project suite. Further on come the parameters:
  - The `/run` parameter tells TestComplete that test or set of tests has to be launched, and not simply open the IDE.
  - The `/project` parameter expresses namely what should be launched (in the given case, the whole of the project, that is, all the Test Items of the project). Via the semi-column, the name of the project is to be written down (in our case, this is `Chapter4`).

3. Now, let's launch to execute a separate function. To do this, we need to run the following command from a command line:

```
"C:\Program Files\SmartBear\TestComplete 9\Bin\TestComplete.exe" "z:\TestCompleteCookBook\TestCompleteCookBook.pjs" /run /project:Chapter4 /Unit:Unit1 /routine:testDemoTestItems2
```

4. In this example, apart from the already known parameters, two more have been added:
  - `/Unit`: This signifies the unit in which the function for the launch is located
  - `/routine`: This signifies the name of the specific function that should be launched



### How it works...

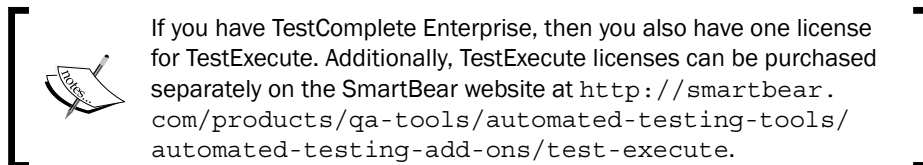
The first method of launching allows the launch of all the enabled tests from the Test Items list. The second method allows to launch a single function. Certainly, we can create just a single function that nests calls of several functions within it, thus covering the whole of the testing plan. Nonetheless, in this case, contingent upon TestComplete getting accidentally hung-up, the possibility of losing entire log entries is much likelier, since Test Items are not used. At the point of using Test Items, the contents of the TestComplete memory, related to the logs, is committed to HDD upon completion of each of the tests.

Another inconvenience lies in the fact that we would have to comment all of the function calls of the corresponding functions to disable specific tests in the second case. Therefore, in the event of using Test Items, it will be sufficient to uncheck the corresponding Test Items.

### There's more...

In TestComplete, there is a variety of additional parameters, allowing us in one way or another to influence the tests launching procedure: disabling the splash screen, *silent* mode (in which tests execution will not be stopped even in the event of a critical error), exiting upon completion of scripts processing (to continue batch-file execution), exiting codes (useful at the point of integration with the systems of continuous integration), and so on.

For running tests from command line you can use a special command-line tool called TestExecute. It uses the same parameters as TestComplete, but doesn't have IDE and can be used only for running tests.



### See also

Running scripts from command line is usually used for creating regular runs:

- ▶ The *Creating test plan for regular runs* recipe



We can also pass additional parameters to script from command line:

- ▶ The *Passing additional parameters to test from the command line* recipe
- ▶ All these parameters could be read up on in greater detail at this link:  
<http://support.smartbear.com/viewarticle/33353/>.

## Passing additional parameters to test from the command line

At the point of launching tests from the command line, there may arise a need to pass some information to be further used in the tests. In another case, while the script being executed, there may turn up a necessity to find out which parameters the TestComplete has been started with that serves the basis for decision-making. In this recipe, we will deal with the methods of working with parameters from the command line, which have been assigned at the point of starting the TestComplete application.

### Getting ready

Launch the TestComplete application with the `/ns` parameter (this parameter blocks display of the so-called splash screen—the picture that appears on the screen right after the application has been launched—until the main window of the application is successfully opened).

### How to do it...

In order to work with command-line parameters, we need to perform the following steps:

1. This simple script will make consecutive log entries of all the command-line parameters of TestComplete:

```
for(var i = 0; i <= BuiltIn.ParamCount(); i++)  
{  
    Log.Message(BuiltIn.ParamStr(i));  
}
```

2. In the result of this code, the log will contain two messages: the pathname to the `TestComplete.exe` file and the `/ns` string.

### How it works...

The `BuiltIn.ParamCount` method gives us the count of the parameters, which have been passed to TestComplete, while the `BuiltIn.ParamStr` method allows obtaining a separate parameter, by assigning its index.

Pay attention to the next peculiarity of these methods: zero-indexed element of the `BuiltIn.ParamStr` method contains the pathname to the executed `TestComplete` file, however, the `BuiltIn.ParamCount` method does not take into account this zero-indexed element and returns only the count of the parameters.

### There's more...

Parameters of any of the process can be obtained with the help of the `CommandLine` property. For instance, in the following example, we receive the entire command line of the `TestComplete` parameter:

```
var cmdLine = Sys.Process ("TestComplete").CommandLine;
```

Since the `CommandLine` property contains the whole of the command line, working with the parameters is not convenient, which is why it is good to write up a function that would return an array of parameters to be accessed, analogous to the `BuiltIn.ParamStr` method at work.

At the point of processing the line we should account for the path to the file, any other parameter may be enclosed in quotation marks and contain blank spaces. This is why the logic of the function will be to the following effect:

- ▶ We consecutively obtain command-line symbols.
- ▶ If the received symbol is not a blank space or a double quotation mark, we simply add the symbol to the parameter-to-be.
- ▶ If this is a double quotation mark, we set or disannul a `Boolean` flag for substring detection.
- ▶ If this is a blank space, first, we are supposed to check if we are inside a substring. If this is so, then we add the blank space to the parameter-to-be. If not, we add the current parameter to the array.

In the result, the function for parsing the command line will appear to the following effect:

```
function parseCommandLine(cmdLine)
{
    var chr, parameter = "";
    var parameters = [];
    var isString = false;

    for(var i = 0; i < cmdLine.length; i++)
    {
```

```
chr = cmdLine.substr(i, 1);
switch(chr)
{
    case '"':
        isString = !isString;
        break;
    case ' ':
        if(isString)
        {
            parameter += chr;
        }
        else
        {
            parameters.push(parameter);
            parameter = "";
        }
        break;
    default:
        parameter += chr;
        break;
}
}
if(parameter != "")
{
    parameters.push(parameter);
}
return parameters;
}
```

And an example of the script, which employs the function in view:

```
function RunTests()
{
    var arr =
    parseCommandLine(Sys.Process("TestComplete").CommandLine);
    for(var i = 0; i < arr.length; i++)
    {
        Log.Message(arr[i]);
    }
}
```

This example outputs all parameters of `TestComplete` to the test log.

## Organizing test plan runs

Tests are useful to group in such a manner that there is a possibility to launch them not only one-by-one or all at once, but also in re-grouped batches. Such a re-grouping may be useful in case there is no necessity to go through the whole of the project suite, and when it is sufficient to test a specific functionality (for example, at the point of delivering a hot fix). The Test Items page, dealt with earlier in the *Creating a test plan for regular runs* recipe, allows us to create a similar structure with the help of groups.

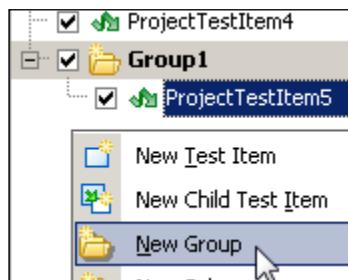
### Getting ready

Create several functions, which we will be able to add to the list of Test Items.

### How to do it...

In order to group the tests, we need to perform the following steps:

1. Open the Test Items panel by right-clicking on project name, select **Edit | Test Items**.
2. Right-click on the list of Test Items and opt for the **New Group** option. In the result, we will have a group with the title **Group1**.

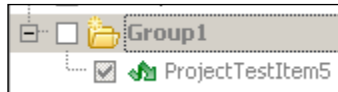


3. Rename the group **Group1** according to the name of the functionality that will be tested by the tests from this group (for example, for Calculator, this could be a group of TestMemory, which will comprise all the tests meant to check the memory functionality). To rename, it is enough to click once on the name of the group and enter a new name.
4. Right-click on the created group and select the **New Child Test Item** option. In the result, the group will have a new Test Item added, which we will handle just like we handled the ordinary Test Items (bound it to the function, set the timeout, and so on).

## How it works...

Having created several groups like that (each corresponding to a specific functionality), we can enable and disable the tests in batch as groups, which will significantly simplify our work. At the end, we need to click on the checkbox to the left-hand side of the group name, and in the result, the status of all the tests in the group will change (it will become enabled or disabled).

Pay attention to the fact that when in the disabled state, the flags do not have an ordinary appearance of the unchecked checkboxes, but rather appear as grey disabled controls.



This happens because the sibling Test Items have three inherent states: enabled, disabled by group, and disabled completely. Completely disabled Test Items (white squares) cannot be enabled even if the entire group is enabled. This may be useful in case a test is presently under construction, or is being modified, as launching such Tests is untenable.

## Scheduling automatic runs at nighttime

When working with a hefty backlog of automated tests, launching those may be long-winded and take hours. One of the effective methods to resolve this problem is automatic launch of the tests when there is no one at work, and tests are not in anyone's way. This could be nighttime, weekends, or holidays. In this recipe, we will consider automatic launch of the tests every night.

## Getting ready

In order to learn how to launch the executable tests from the command line, read the *Running tests from the command line* recipe.

## How to do it...

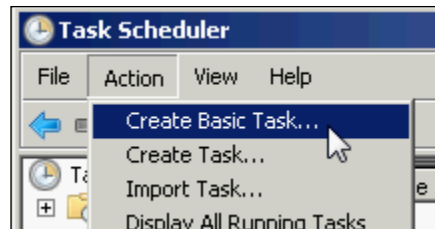
In order to schedule test run, we need to perform the following steps:

1. Create the `starttc.bat` file with the following entrails:

```
taskkill /f /t /im testcomplete.exe
timeout /T 10
set TCPATH=C:\Program Files\SmartBear\TestComplete 9\Bin
"%TCPATH%\TestComplete.exe" Z:\TestCompleteCookBook\
TestCompleteCookBook.pjs /run /project:Chapter4 /exit
```

we will be launching this file by the schedule.

2. Launch Windows Task Scheduler by going to **Start | All Programs | Accessories | System Tools | Task Scheduler**.
3. In the **Task Scheduler** window, select **Action | Create Basic Task...**



4. In the appearing window, **Create Basic Task Wizard**, enter the name of the task (for example, `TestComplete Tests`), and click on the **Next** button.
5. In the **Task Trigger** window, select the **Daily** option, and click on **Next**.
6. In the next window, signify the date and time for the first launch (the date may be left at present, the time as `00:00:00`), and then click on **Next**.
7. In the **Action** window, select the **Start a program** option, and click on **Next**.
8. Click on the **Browse...** button and select the created `starttc.bat` file.
9. Click on **Next**.
10. In the **Summary** window, check all the entered data is correct or not, and click on **Finish**.
11. In the left-hand side part of the **Task Scheduler** window, select the **Task Scheduler Library** element.
12. In the right-hand side part of the **Task Scheduler** window, locate the newly created task `TestComplete Tests`, and right-click thereon to further select the **Run** option.

If everything has been properly done, the tests will be automatically launched for execution.

This task will be launched every night.

### How it works...

Since we cannot launch more than one copy of TestComplete, the first and foremost thing to do in the `.bat` file is closing the respective process thread with the help of the `taskkill` command, which is a part of Windows. This is done despite the possibility of TestComplete remaining open for some reason at the moment of the task's launch.

Further on, with the help of the standard timeout command, we await for 10 seconds. It is necessary to have all the resources duly released that are being utilized by TestComplete, or else upon successive launch we could run amuck with error messages. Depending on the speed of the computer, this parameter may be greater or lesser, nonetheless we recommend it be left at a greater value (for example, 30 seconds). Such an insignificant lag will have no impact on the timing of the test's execution, rather this parameter will not have to be customized each time against launches being transferred onto another computer.

Further more, with the help of the `set` command, we create a `TCPATH` variable, which will contain the pathway to the folder wherein TestComplete resides. This is optional, however the pathway can be signified fully in the following command, but then the command line could be long-drawn-out. In the last command, we launch the tests from the `Chapter4` project up to execution. With the help of `/exit` parameter, we signify that, having completed all the tests, TestComplete should close down.

### See also

Running tests from command line is explained in detail in the following recipe:

- ▶ *The Running tests from the command line recipe*

## Running tests via Remote Desktop

If you launch tests remotely via **Remote Desktop** connectivity to your computer, you should follow two important rules:

- ▶ Do not minimize the Remote Desktop window
- ▶ Do not close the Remote Desktop window

If any of the above mentioned rules are not followed, the system ceases redrawing GUI elements on the remote computer and TestComplete will not be able to continue working with the controls elements (this is not the TestComplete-specific problem, it is OS-related and will reproduce for all automation tools that work with applications via GUI). This does not mean that it is necessary to incessantly keep the window open over the whole screen upfront. It is possible to change the sizing of the Remote Desktop window down to its minimum (which will make the screen of the remote computer invisible) and place it onto the *back burner* against the background to keep it out of your way for the work at hand. At this time, you can simultaneously run any actions on the local computer, which will not cramp up the tests on the remote computer. If even this doesn't work then you can customize the system so that the Remote Desktop window can be minimized out of view.

## How to do it...

To be able to minimize the Remote Desktop window, the following steps should be performed on a local computer which is used to connect to the remote PC:

1. Click on **Start**, in the **Search** box, write in `regedit`, and press *Enter*. The **Registry Editor** window will open up.
2. In the left-hand side part of the window, navigate to **HKEY\_CURRENT\_USER | Software | Microsoft | Terminal Server Client**.
3. Select the **Edit | New | DWORD (32-bit) Value** option. In the right-hand side part of the window, new value will be assigned.
4. Rename it to `RemoteDesktop_SuppressWhenMinimized`.
5. Double-click on the newly created element and in the **Value data** field, input the value `2`.
6. Repeat the steps 3 to 5 by navigating to the following element: **HKEY\_LOCAL\_MACHINE | SOFTWARE | Microsoft | Terminal Server Client**.
7. Now you can go ahead and safely minimize the Remote Desktop window, however, it should not be closed anyways.

## How it works...

Closing the Remote Desktop window is tantamount to blocking the computer (which occurs when pressing this key combination: *Ctrl + Alt + Delete*).

Since the system stops redrawing GUI element upon being blocked, `TestComplete` cannot interact with the tested application.

## Changing playback options

Parameters of playback of the tested scripts can be customized with the help of the Playback options page.

In this recipe, we will consider some of the most interesting options and the way they influence the performance of the scripts.



## How to do it...

In order to change playback options, perform the following steps:

1. In the TestComplete application, right-click on the name of the project, and go for the **Edit | Properties** option.
2. In the opened project panel, on left-hand side, select the **Playback** element.
3. In the result, the **Playback** properties page will open up. Here you can customize the playback parameters of the tested scripts.

## How it works...

- ▶ **Stop on error, Stop on warning, and Stop on window recognition error:** These options allow stopping scripts execution contingent upon any of the signified events having occurred. It is usually recommend one should disable these options, since they influence all the tests.
- ▶ **Minimize TestComplete:** This option minimizes the main **TestComplete** window for the time being while tests are being executed, which prevents the possibility that the **TestComplete** window might appear on the foreground instead of in the window of the tested application.
- ▶ **Disable mouse:** This option disables the possibility of a user using a mouse, and thus impeding with the test's results. This is especially recommended when employing low-level procedures.
- ▶ **Auto-wait timeout:** This is the waiting time for the windows and controls elements that mark their failure to appear. This parameter should be increased at the time of transference of testing to a slower computer.
- ▶ **Delay between events, Key pressing delay, Dragging delay, and Mouse movement delay:** These options allow changing the delay between corresponding events—the data input from the keyboard, mouse-clicks, and operations with windows. Increasing these parameters is recommended when errors related to the speed of execution occur during scripts playback (for example, admission rate of typed symbols during text input, dragging-and-dropping incorrect objects, and so forth.). Another possibility can be timing slowdown to visually monitor all the actions and pinpoint the reason for the mistake.
- ▶ **On unexpected window:** Options from this group allow assigning TestComplete behavior against the appearance of a window that will hinder the flow of test execution (for example, a modal window with an error). In the meantime, the log will contain an error and the actions that have been undertaken for the appeared window.

- ▶ **Ignore overlapping window:** This option allows the program to ignore the overlapping windows and controls elements. These are GUI elements that overshadow the controls elements, with which TestComplete is currently working, however, it does not block the tested application. Often, it is worthwhile to switch on this option, if the log contains the *Overlapping Window* error, however, visually nothing stands in the way.
- ▶ **Post image on error:** This is a very useful option, allowing automatically placing a screenshot in the event of emergent error. This option is preferable to be switched on all the time, since the screenshot will help to detect the underlying cause of the problem.
- ▶ **Save log every N minutes:** This option stores the log to the disk within the preassigned interval, preventing its loss in critical cases such as blackout, emergency shutdown of the TestComplete application, and so on.
- ▶ **Store last N events:** This option allows to specify the minimum number of events to be entered to the log. Usually the events logs take up the largest portion of the log, although there is no need to keep them all. In the case that this option has been set and an error has occurred, only the specified number of events will be stored in the log.

## Increasing run speed

If your tests are being executed slowly, it may be down to a number of factors. In this recipe, we will consider the most frequently emerging problems and their possible resolutions.

### How to do it...

In order to increase run speed, we can perform the following changes:

1. Disable Visualizer (detailed steps can be found here: <http://support.smartbear.com/viewarticle/28968/#Enabling>).
2. Use the filters in the **Object Browser** tab to decrease number of the objects displayed.
3. Work directly with the properties (an example can be found in the *Entering text into text fields* recipe in *Chapter 3, Scripting*).
4. Disable the unused TestComplete plugins by going to **File | Install Extension...** menu and unchecking the unnecessary plugins.
5. Disable Debug Agent (detailed information can be found here: <http://support.smartbear.com/viewarticle/27468/>).

## How it works...

Visualizer allows us to capture screenshots of the window that TestComplete is currently working with each time TestComplete interacts with the controls elements. Since this can happen quite often, creation of the screenshots will be time consuming, and there is no necessity to capture all the resulting screenshots, so, it is recommended that the Visualizer be turned off. The method to turn off the Visualizer varies in different versions of TestComplete.

Frequent resorting to the `aqUtils.Delay` method in the tests may significantly slow down scripts execution time. Try to avoid usage of this method by replacing it with the corresponding `wait` functions.

The **Object Browser** tab allows us to flexibly customize the displayed objects with the help of the filters buttons in the upper part of the **Object Browser** panel. It is recommended to disable the invisible objects and display only the processes that pertain to the tested applications. This will expedite renewal of the tree of objects.

## See also

- ▶ The *Waiting for an object to appear* recipe in *Chapter 5, Accessing Windows, Controls, and Properties*
- ▶ The *Entering text into text fields* recipe in *Chapter 3, Scripting*

## Disabling a screensaver when running scripts

During the time the scripts are being executed, at the most malapropos moment, the screensaver may kick in or the computer may go into the sleep mode. The simplest way to avoid such mishaps is to completely turn off these working modes manually. In this recipe, we will consider a way to do so programmatically, if switching those modes manually is not acceptable to you by any reason.

## How to do it...

To programmatically disable the monitor, we need to perform the following steps:

1. To disable the screensaver, it is enough to evoke the API-function `SystemParametersInfo`, passing it the necessary parameters:

```
Win32API.SystemParametersInfo  
(Win32API.SPI_SETSCREENSAVEACTIVE, false, null, 0);
```

2. Disabling the sleep mode of your computer and preventing the monitor snooze is a little more complicated and requires calling the `SetThreadExecutionState` function from the `kernel32.dll` file within the predefined time intervals. The function itself will appear as follows:

```
function disableSleep()
{
    var ES_SYSTEM_REQUIRED = 0x00000001;
    var ES_DISPLAY_REQUIRED = 0x00000002;

    var kernel = DLL.DefineDLL("kernel32");
    var lib =
    DLL.Load("c:\\Windows\\System32\\kernel32.dll");
    var proc = kernel.DefineProc("SetThreadExecutionState",
    vt_i2, vt_void);
    lib.SetThreadExecutionState(ES_SYSTEM_REQUIRED |
    ES_DISPLAY_REQUIRED);
}
```

3. The function call in the script will be made in the following manner:

```
Utils.Timers.Add(30000, "Unit1.disableSleep", true);
```

Here `Unit1` is the name of the unit, in which the `disableSleep` function has been defined.

## How it works...

With the help of the `Utils.Timers.Add` method, we create a timer that will launch a specific function (in our case, `disableSleep` from the `Unit1` unit) within the set interval (in our case, within 30 seconds).

The `disableSleep` function itself will include the dynamic `kernel32.dll` library and call the predefined `SetThreadExecutionState` function. We pass into this function the only parameter that contains the following two flags:

- ▶ `ES_SYSTEM_REQUIRED`: This flag requires the system should work in the normal working mode, preventing sleep or snoozing inactivity.
- ▶ `ES_DISPLAY_REQUIRED`: This flag prevents monitor fading.

Note that disabling the screensaver makes permanent system changes, while preventing the monitor fading works only when scripts run.

## Sending messages to Indicator

**Indicator** is a small window that is displayed in the upper-right corner of the screen when the scripts are being executed which displays the information about the processes taking place at the given moment of time. With the help of the `Indicator` object, we can display our messages in this window.

### How to do it...

In order to display messages in the **Indicator** window we need to perform the following steps:

1. The following script demonstrates basic possibilities of work with **Indicator**:

```
function testIndicator()
{
    Indicator.PushText("Waiting for ABCD process...");
    Log.Message(Indicator.Text);
    if(!Sys.WaitProcess("ABCD", 10000).Exists)
    {
        Log.Message("Process not found");
    }
    Indicator.Clear();
    aqUtils.Delay(5000, "Waiting 5 seconds...");
    Indicator.Hide();
    aqUtils.Delay(5000, "Waiting 5 more seconds...");
    Indicator.Show();
}
```

2. This function will first show the `Waiting for ABCD process...` message and put the message from the `Indicator` object to log.
3. Then `Indicator.Text` will be cleared and a new message `Waiting 5 seconds...` will be shown.
4. The last message sent to the `Indicator` object, `Waiting 5 more seconds...`, will not be visible since we hide the whole `Indicator` object before posting it.

### How it works...

With the help of the `PushText` method, we place the text into the `Indicator` object that is possible to be read via the `Text` property. The `Clear` method clears the text in the `Indicator` object. The `Hide` and `Show` methods allow hiding and displaying anew the `Indicator` object on the screen.

If you would like to create a delay in the script with the help of the `agUtils.Delay` method, you can also place an arbitrary text into the `Indicator` object, having passed it through the second parameter to the `Delay` method.

### There's more...

The `PushText` method does not simply place a new text into the `Indicator` object, but rather retains the previously inputted text. Subsequently, the obsolescent text can be restored with the help of the `PopText` method.

## Showing a message window during a script run

Sometimes it is necessary to interrupt the script being executed and display a certain message to the user, and then continue script execution only on condition the user closes the message.

Note however, that interaction of the automated tests with the user is considered bad style in testing automation, however, there could be instances when this cannot be avoided. For example, if we are running a lengthy test with a lot of actions, we may require that something be done manually in the middle of the test execution (for example, connect a device or reboot a remote server).

### How to do it...

1. To display the message and waiting for the coming interaction on behalf of the user, the `BuiltIn.ShowMessage` method is to be applied.
2. In the following example its use is demonstrated:

```
function testShowMessage()
{
    Log.Message("This code will run before showing a
message.");
    BuiltIn.ShowMessage("Press OK to continue execution...");
    Log.Message("This code will run after message has been
closed");
}
```

### How it works...

The `BuiltIn.ShowMessage` method disrupts text execution and waits till the user clicks on the **OK** button. Meanwhile, access to other `TestComplete` possibilities is shut off. Test execution is expected to continue after the window has been closed.

### There's more...

In TestComplete there is more advanced feature for user interactions—**User Forms**. With the help of User Forms one can create complex dialog windows with various controls elements. In fact, it is possible to create a small application within the TestComplete application. These forms can be useful if, before launching the tests, we would like to assign testing parameters (for example, address of the server, user name, tested application version, and any other customizable parameters).

If you would like, however, to have the possibility of launching tests in the automated mode without human interactions (for example, at night, on the weekend, or each time after the application has been compiled), these parameters are best stored in a configuration file.

# 5

## **Accessing Windows, Controls, and Properties**

In this chapter we will cover the following recipes:

- ▶ Choosing Object Tree Model
- ▶ Understanding the window's life cycle
- ▶ Ignoring overlapping windows
- ▶ Dragging one object into another
- ▶ Calling methods asynchronously
- ▶ Verifying if an object has a specific property
- ▶ Finding objects by properties' values
- ▶ Waiting for an object to appear
- ▶ Waiting for a property value
- ▶ Mapping custom control classes to standard ones
- ▶ Using text recognition to access text from nonstandard controls
- ▶ Using Optical Character Recognition (OCR)
- ▶ Dealing with self-drawn controls not supported by TestComplete



## Introduction

In order to efficiently work with applications under test, we need to know how TestComplete interacts with tested applications, namely with windows and objects within the windows.

In this chapter, we will cover different ways of interacting with controls and learn how TestComplete works with them.

## Choosing Object Tree Model

Object Tree Model is a very important project setting, which influences object recognition in the **Object Browser**. Selection of just one of the models would not make it possible to simply switch between models in the future, keeping the existing scripts; since these models are altogether different. In this recipe we will consider two supported Flat and Tree models, their pros and cons, and provide counsel on the choice of a certain model for your project.

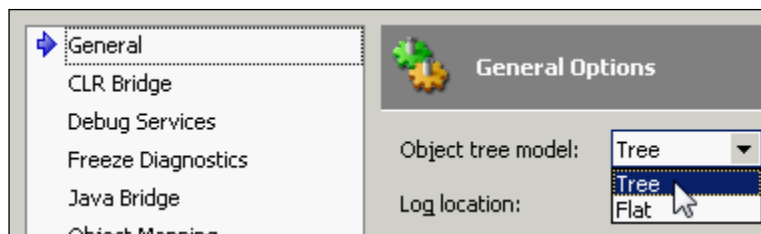
### Getting ready

Launch a standard application Paint from the Windows suite by navigating to **Start | All Programs | Accessories | Paint**.

### How to do it...

To see the difference between two models, we need to perform the following steps:

1. Open the properties of the project (right-click on the name of the project and navigate to **Edit | Properties**).
2. Opt for the **General** group of options.
3. Set the **Object tree model** parameter value to **Tree**, as shown in the following screenshot:



4. Begin recording of the script and click on any button on the **Home** panel in the Paint window, and then stop the recording.

5. If, in the meantime, the **NameMapping** window appears, click on the **Cancel** button.
6. In the result, the following script string will be recorded:

```
Sys.Process("mspaint").Window("MSPaintApp",
    "**").Window("UIRibbonCommandBarDock",
    "UIRibbonDockTop").Window("UIRibbonCommandBar",
    "Ribbon").Window("UIRibbonWorkPane",
    "Ribbon").Window("NUIPane").
    Window("NetUIHWND").Click(260, 78);
```

7. Reiterate steps 1 through 4, however, on the first step, set the **Object tree model** parameter value to **Flat**.
8. In the result, the following code will be recorded:

```
Sys.Process("mspaint").Window("MSPaintApp",
    "**").Window("NetUIHWND").Click(260, 78);
```

### How it works...

The difference between the Flat and the Tree models is clearly distinguishable; the Tree model yields a much greater number of nested objects as compared to the Flat one. This occurs due to the Tree **Object Browser** mode displaying controls elements in exactly the same way as they are located in the tested application itself. In the Flat model, all the controls elements are sibling elements to the window, regardless of their internal mapping in the application. As the toolbar, usable in Paint (Ribbon), is quite a complicated control element, we observe a complex element structure in the Tree model between the window and the toolbar; there are to be found as many as four elements. Naturally, the Flat model seems more user friendly, and it is indeed so. If you are not running up against a difficulty in naming the controls elements in this mode, this is to be preferred as your first option.

Sometimes, however, there arise use cases that make it difficult to apply the Flat model. For example, if there are too many eponymous controls elements in the window that are instanced to the same classes, then all of them are outputted equally on the same level of the hierarchy (that is, as sibling elements in relation to the window they have taken after); therefore, it becomes quite hard to sort out where each and every controls element exactly belongs. The only way of extrication consists in using the indices (which is not really a convenient thing). This is why the thumb rule for choosing Object Tree Model runs as follows: for simple applications where you don't have problems with recognizing controls, use Flat model. In other cases use Tree model.

If you use the Tree model, it is recommended that you should apply the controls element of the NameMapping project item and its extension of Aliases. NameMapping allows us to assign the controls elements with simple and brief names, while the Aliases allows us to create a hierarchy from these named objects (regardless of the way they have been mapped in the application and in the **Object Browser**).

## See also

- ▶ If you are applying the Flat model and often come up against an issue of elements recognition with the same names, it is recommended that you revert to the *Finding objects by properties' values* recipe
- ▶ If you test web applications, we recommend reading the *Choosing Web Tree Model* recipe in *Chapter 10, Testing Web Applications*

## Understanding the window's life cycle

For the sake of code decomposition towards simplification, we create variables that correspond to the objects (processes, windows, and controls elements). In this recipe, we are about to deal with correct appropriation of such variables in TestComplete.

## Getting ready

Add Calculator Plus to **TestedApps** (C:\Program Files\Microsoft Calculator Plus\CalcPlus.exe).

## How to do it...

In order to understand the life cycle of windows we will perform the following steps:

1. Launch the following function to be executed:

```
function testVariables()
{
    TestedApps.CalcPlus.Run();
    var wCalc = Sys.Process("CalcPlus").Window("SciCalc",
        "Calculator Plus");
    wCalc.Activate();
    wCalc.Close();
    aqUtils.Delay(2000);
    TestedApps.CalcPlus.Run();
    wCalc.Activate();
}
```

2. In the result, we will get the error message **The window was destroyed during method execution.**

## How it works...

We launch the calculator and create a `wCalc` variable corresponding to the main window of the calculator. Then we close the calculator and reopen it again in an attempt to activate the same using the pre-existing variable `wCalc`.

At first, everything should work without a hitch, because in both instances access to the main window takes place in seemingly the same way: the application has been launched and appears just like it did the first time; nonetheless, we are in receipt of the error message.

This is happening because the `wCalc` variable has been assigned to an object which is no longer in existence as of the moment the program was closed down with the help of the `Close` method. If a new object has the same name, it does not mean it is the same object. The variable `wCalc` still tries working with the older window, which has been closed by now, and will never reinstate. When the variable is being addressed, `TestComplete` does not recalculate the expression, which we assigned to the variable in the beginning.

If we attempt to use the variable `wCalc` again to work with the newer window, we need to initialize it again with the same expression as done initially:

```
wCalc = Sys.Process("CalcPlus").Window("SciCalc", "Calculator  
Plus");
```

For brevity's sake, the right-hand part of the expression can be made into a function, which returns the window object.

This example is quite a vivid one; however, in our tested applications, there may arise less obvious and underhanded situations. For example, the window remains intact, while the elements inside are being renewed. Such a peculiarity should always be kept in the back of your mind if you are using variables to handle screen (that is, display) objects.

## Ignoring overlapping windows

Sometimes, at the point of scripts being executed in an application at hand, there are windows or other controls elements being simultaneously opened to overlap the controls elements that are being referred to by the scripts, at the given moment of time. As a result, in the log, there appear error messages.

**Overlapping window** is the window that overrides the controls element, which `TestComplete` is handling at the moment; however, it does not block the working with the controls element.

In this recipe, we will learn to rid ourselves of the previous-mentioned error in the simplest manner imaginable.

## Getting ready

Launch a standard Windows Notepad application (C:\Windows\notepad.exe).

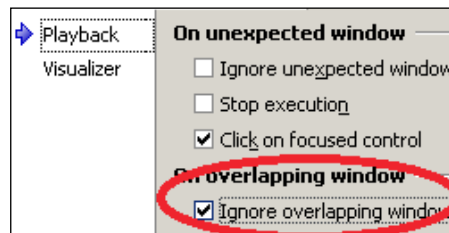
## How to do it...

In order to deal with overlapping windows we need to perform the following steps:

1. Create the following function:

```
function testOverlappingWindow()
{
    var wNotepad = Sys.Process("NOTEPAD").Window("Notepad",
        "*");
    edit = wNotepad.Window("Edit");
    edit.Keys("first string[Enter]");
    wNotepad.MainMenu.Click("Edit|Find...");
    edit.Click(100, 120);
    edit.Keys("second string");
    var wFind = Sys.Process("NOTEPAD").Window("#32770",
        "Find");
    wFind.Window("Button", "Cancel").ClickButton();
}
```

2. In the line `edit.Click(100, 120);` change the parameters that are being passed to the method so that the mouse-click would be made at a point close by the **Find** window (you might need several tries to find the correct coordinates). Launch the function.
3. In the result, we will have the following error message in the log: **There was an attempt to perform an action at point (802, 545), which is overlapped by another window.**
4. Now, open the properties of the project (right-click on the name of the project and navigate to **Edit | Properties**).
5. Open the **Playback** group of settings.
6. Enable the **Ignore overlapping window** option as shown in the following screenshot:



7. Launch the function again. This time, there are no errors in the log.

## How it works...

First we try to run the function which clicks on an object which is hidden by another window. As a result, an error message is put to the log showing that overlapping window prevents TestComplete performing the click action.

After we had made changes to project options, the error wasn't sent to the log again.

The example with the **Find** window in Notepad shows that error with overlapping windows isn't critical, since the script keeps successfully on after posting an error. Another example of such a behavior is an invisible controls element, which is located above the targeted element in mind, however, not standing in the script's way. Since such messages in the log may hinder us, we can simply ignore them, as shown in this recipe.

## There's more...

If the window is blocking script workings with the application, it is called **Unexpected**, not **Overlapping**. In **Playback** group of project settings, one can customize behavior of TestComplete at work with the same windows, and we will also get to grip with the way to process those from the scripts in *Chapter 12, Events Handling*.

## See also

- ▶ If you don't want to ignore all overlapping windows, but only some of them avail yourself of the *Disabling certain error messages* recipe in *Chapter 12, Events Handling*
- ▶ To deal with Unexpected windows refer to the *Handling unexpected windows, part 1* and *Handling unexpected windows, part 2* recipes in *Chapter 12, Event Handling*

## Dragging one object into another

With the help of the `Drag` method in TestComplete, we are able to drag-and-drop any object onto the predefined location in terms of pixels vertically and horizontally (on condition, of course, that the object supports the drag-and-drop behavior); however, there is no built-in possibility to drag one object onto or over another.

We can create a function to tackle such a feat by using coordinates of the objects. The task at hand is writing a script that would drag the **Find** window in the Notepad and place the same on the center of the main window, regardless of its dimensions and initial whereabouts.

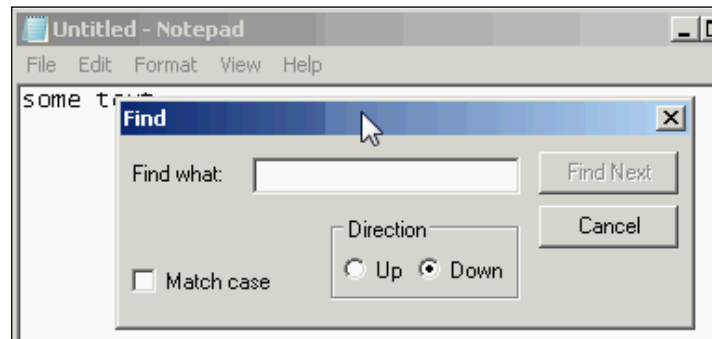
## Getting ready

Launch the Notepad application (C:\Windows\notepad.exe).

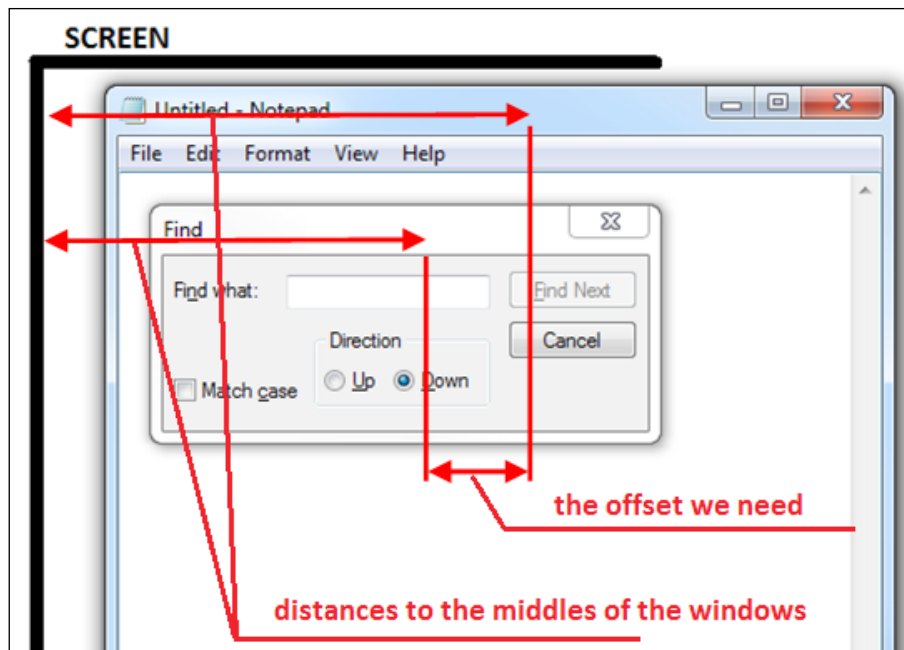
## How to do it...

The process of creating a function for dragging a window consists of the following steps:

1. The `Drag` method takes four parameters: coordinates inside the window on which the mouse-click is expected to be made at the point of dragging (`X` and `Y`) and offset coordinates by width and height for dropping (`offsetX` and `offsetY`). In our case, the object that is being dragged-and-dropped is the **Find** window which is shown in the following screenshot:



2. The `X` parameter inside the window, would be set equal to half the width of the **Find** window, while the `Y` parameter would be equal to 10 (the height of the heading of the window, tantamount to 21 pixels by default; so target coordinate will be in the middle of the window caption).
3. The offset parameters should be calculated, taking the dimensions of both of the windows into account (the main **Notepad** window and the **Find** window).
4. To figure out the distance along the horizontal line at which the window is to be relocated on the screen, we need to obtain the coordinates of the center of the main window, and add to it the distance from the border of the window down to that of the screen. Then we subtract the difference from the border of the **Find** window down to the beginning of the screen, subtracting half the width of the **Find** window (as we will be dragging the **Find** window by its middle, as the pivotal point), as shown in the following screenshot:



5. Similarly, the vertical offset is to be calculated by using the height of the window and the vertical distance.



If you have a hard time understanding the logic of the calculations, you may feel lead to draw schematic view of the location of the windows on the screen, and thus, shape up the formula by cracking numbers, and then transform those numbers into the meaningful variables.

6. As a result of this, we can write up the following function. This function would open the Notepad and maximizes it over the whole screen, then opens the **Find** window and drags it to relocate onto the center of the main window. After this, the function restores the original size of the main window, and again, places the **Find** window on the center.

```
function testDragDrop()
{
    var pNotepad = Sys.Process("notepad");
    var wNotepad = pNotepad.Window("Notepad");
    wNotepad.Activate();
    wNotepad.Keys("Some text");
    wNotepad.Maximize();
    wNotepad.MainMenu.Click("Edit|Find...");
    var wFind = pNotepad.Window("*", "Find");
```



```
var offsetX = wNotepad.Width/2 + wNotepad.ScreenLeft -
    wFind.ScreenLeft - wFind.Width/2;
var offsetY = wNotepad.Height/2 + wNotepad.ScreenTop -
    wFind.ScreenTop - wFind.Height/2;
wFind.Drag(wFind.Width/2, 10, offsetX, offsetY);

wNotepad.Restore();
offsetX = wNotepad.Width/2 + wNotepad.ScreenLeft -
    wFind.ScreenLeft - wFind.Width/2;
offsetY = wNotepad.Height/2 + wNotepad.ScreenTop -
    wFind.ScreenTop - wFind.Height/2;
wFind.Drag(wFind.Width/2, 10, offsetX, offsetY);

wFind.Close();
}
```

### How it works...

With the help of the `Width`, `Height`, `ScreenLeft`, and `ScreenTop` properties of both of the controls elements (in our use case, we are talking about windows), we calculate the distance at which it is necessary to relocate the object to have it appear over the center of another object in play.

As seen from this example, the code for calculation of the coordinates is re-iterated, and this is why it is better to extract it into a separate function. The function would accept two parameters: the object that is being dragged, and the pivotal object onto which the drag-and-drop is targeted.

### There's more...

Here is an example of universal function which can be used for dragging objects:

```
function dragObject(objDrag, objTo)
{
    var offsetX = objTo.Width/2 + objTo.ScreenLeft -
        objDrag.ScreenLeft - objDrag.Width/2;
    var offsetY = objTo.Height/2 + objTo.ScreenTop -
        objDrag.ScreenTop - objDrag.Height/2;
    objDrag.Drag(objDrag.Width/2, 10, offsetX, offsetY);
}
```

Our initial code could be simplified by replacing the blocks from the three lines of code, which calculate coordinates and evoke the `Drag` method, with the following singular function:

```
dragObject(wFind, wNotepad);
```



The Y parameter has been hard-coded as being equal to 10, since we have been at that point in the window. To drag other objects, it makes sense to replace this value for half the height of the object that is being dragged (`objDrag.Height/2`), so that mouse-click would always target the center of the object.

## Calling methods asynchronously

Sometimes, there arises a necessity to launch some by-process in the tested application (for example, some long-drawn mathematical calculation), and to simultaneously carry out other actions, unhindered. In this recipe we will consider how this can be achieved.

### Getting ready

Launch the Calculator Plus (`C:\Program Files\Microsoft Calculator Plus\CalcPlus.exe`) and Notepad (`C:\Windows\notepad.exe`) applications.

### How to do it...

We need to perform the following steps:

1. The following function launches the calculator for factorial calculation of a hefty number. This takes up a long time, during which the calculator is unavailable for the user. Let's suppose a user has to fulfill certain operations in the meantime, and that in the Notepad.

```
function testAsynch()
{
    var wCalc = Sys.Process("CalcPlus").Window("SciCalc",
"Calculator Plus");
    wCalc.Keys("123321123321");
    var button = wCalc.Window("Button", "n!");
    button.Click();
    var wNotepad = Sys.Process("notepad").Window("Notepad");
    wNotepad.Activate();
    wNotepad.Keys("some text");
}
```

2. After calling the `Click` method for the button of the factorial calculation, our function will keep waiting until the calculator is made available again (about 20 seconds, after which a window shows up with a warning). Only afterwards would the Notepad window be enabled with some text inputted into it.

3. Now, let's replace the following string:

```
button.Click();
```

With the next one:

```
Runner.CallObjectMethodAsync(button, "Click");
```

And evoke the function again.

4. At this time, manipulations with the Notepad would occur immediately after clicking on the button for factorial calculations in the calculator.

### How it works...

The `Runner.CallObjectMethodAsync` method accepts two parameters: the `object` and the `callee` method. If the method takes these parameters, they should be passed on to the `CallObjectMethodAsync` method immediately after the name of the evoked method.

In the result, the method of the object will be called asynchronously, and the thread of execution will be relegated to the line of the script that follows.

The `CallObjectMethodAsync` method does not create a new thread of execution, which in turn implies we cannot, as an example, launch the method `aqFile.Copy` asynchronously in order to copy a large file. Use of this method is recommended only for calling private methods of the tested application.

### There's more...

If, upon execution of several actions, we still need to wait for the operation to be completed, having been launched asynchronously, we could resort to the value that is to be returned by the `CallObjectMethodAsync` method:

```
var res = Runner.CallObjectMethodAsync(button, "Click");  
// some actions  
res.WaitForCompletion(20000);
```

The `WaitForCompletion` method has one mandatory parameter: maximal waiting time.

## Verifying if an object has a specific property

If we try addressing an unsupported property of an object's method an error would show up in the log.

To first verify if a property or a method is supported by the given controls element, we can use a specific method of `IsSupported`, that is extended by the `aqObject` object.

In this recipe, we will consider an example of using the `IsSupported` method.

## Getting ready

Launch Calculator Plus (C:\Program Files\Microsoft Calculator Plus\CalcPlus.exe).

## How to do it...

The following function demonstrates usage of the `aqObject.IsSupported` method with the example of the Calculator Plus application:

1. In the loop, we will fine-tooth-comb all the sibling controls elements of the application and have the value of the `wText` property outputted to the log, if the same is available in the given controls element.
2. Write and launch the following function:

```
function testPropertySupported()
{
    var wCalc = Sys.Process("CalcPlus").Window("SciCalc");
    wCalc.Activate();
    var objects = wCalc.FindAllChildren("Visible", true);
    objects = (new VBAArray(objects)).toArray();

    for(var i = 0; i < objects.length; i++)
    {
        if(aqObject.IsSupported(objects[i], "wText"))
        {
            Log.Message(objects[i].wText);
        }
        else
        {
            Log.Warning("Object '" + objects[i].Name + "' doesn't
                have wText property");
        }
    }
}
```

3. In the result of the preceding function call, we will obtain one message with the text **0**. (the text from the result output field of the calculator) and several warnings concerning the property being unsupported.

## How it works...

With the help of the `FindAllChildren` method, we will obtain an array of all the sibling objects of the calculator window. Since the `FindAllChildren` method returns an array suite-formatted after Visual Basic, we need to transmute the same to fit the JScript array format with the help of the `toArray` method.

Further on in the loop, with the help of the `aqObject.IsSupported` method, we check if the `wText` property is available for each controls element, and thus we output the value of this property, provided it exists for the given controls element, else a warning occurs related to the missing property of the `ith` object in the loop.

Property availability validation for an object of a specific method is executed in a similar way.

## See also

- ▶ To learn more about finding objects by property values refer to the *Finding objects by properties' values* recipe

## Finding objects by properties' values

Sometimes, there arises a need to locate an object by a number of characteristic properties (for example, choosing an enabled element from among several look-alike ones). `TestComplete` provides the `Find` method, allowing location of the controls element by several properties and their values.

In this recipe, we will search for and find one of the two buttons with the same text of `C`. There are two such buttons: one works to clear the results input and the other stands for hexadecimal calculations. We need to select and click on the button for clearing the results.

## Getting ready

Launch the Calculator Plus application (`C:\Program Files\Microsoft Calculator Plus\CalcPlus.exe`) and switch it to Scientific mode (navigate to **View | Scientific**).

## How to do it...

The following example demonstrates usage of the `Find` method:

1. First we declare two arrays which contain properties and values of the desired object as follows:

```
var properties = ["WndCaption", "Enabled"];
var values = ["C", true];
```

2. Then we call the `Find` method for the object where we look for the control:

```
var button = wCalc.Find(properties, values);
```

3. This method returns the object which corresponds to given properties.
4. The following function executes search of the targeted button (C) and triggers a mouse-click on it:

```
function testFindControl()
{
    var wCalc = Sys.Process("CalcPlus").Window("SciCalc");
    wCalc.MainMenu.Click("View|Scientific");
    wCalc.Refresh();

    var properties = ["WndCaption", "Enabled"];
    var values = ["C", true];
    var button = wCalc.Find(properties, values);
    button.Click();
}
```

### How it works...

First of all, we select the required mode (Scientific), then update the information about the main calculator window with the help of the `Refresh` method. If this is left undone, `TestComplete` would pick up the obsolete copy of the window, which used to be available in the Standard mode of the calculator.

Then we declare two arrays. The first array (`properties`) contains the names of all the properties, which we will loop through searching for the targeted element, while the second array (`values`) contains values of these properties.

In our case, we are searching for the controls element by screening the text of the button (`WndCaption`) and by looking up its availability (`Enabled`), since the `Enabled` property is different for the two buttons in view.

Further on, with the help of the `Find` method, called for the window of the calculator, we get busy with the search of the targeted controls element. The `Find` method returns the found object, on which we make a mouse-click.

The number of the properties and their values may be arbitrary; however, one should desist from complicating them too much. It is sufficient to write in only the number of the properties that will conclusively pinpoint the object.

## There's more...

We can also pass a third parameter to the method `Find`, namely that of `Depth` (the nesting depth for the searching procedure). By default, this parameter is equal to 1, that is, the search is carried out only amongst the sibling objects of the controls element, targeted by the `Find` method. If the `Depth` parameter is too large (for example, 999), the search would be carried out throughout the whole of the objects tree, beginning with the main element therein.

Apart from the `Find` method, there is another method called `FindChild`. The difference between the `Find` and `FindChild` methods is the fact that the `Find` method checks not only the sibling objects up to matching the pre-assigned search criteria, but also the object itself.

The methods `Find` and `FindChild` return the first controls element that has matched the search criteria. If there is a need to find several controls elements, we could use the methods `FindAll` and `FindAllChildren`, which return all of the matches by pre-assigned properties and values, not just those of one object. In this case, the returned value would be that of an array formatted as `VBAArray`.

In case of JScript, the array in format of `VBAArray` should be type-cast to the format of JScript. This type-case is done via the `VBAArray` object and the `toArray` method. For example:

```
var controls = wCalc.FindAll(properties, values) ;
controls = (new VBAArray(controls)).toArray();
```

## Waiting for an object to appear

Sometimes, there arises a need to pause script execution until a certain window shows up. It usually happens when scripts expect an application to work faster than it actually does.

For example, this would be the case, when the tested application executes time-consuming calculations, and displays the window with the result once these are completed.

To resolve this task in TestComplete there are so called `wait` methods; we will consider the `waitWindow` method as an example.

## Getting ready

For our example, we will launch Calculator Plus (`C:\Program Files\Microsoft Calculator Plus\CalcPlus.exe`) and launch factorial calculation for quite a large number. In case of prolonged operation, the calculator displays a message every 40 seconds to notify that the operation could last quite a long time, suggesting one either continue to wait or stop the calculation.

Make sure, that the calculator has the **Digit grouping** option disabled from the **View** tab.

## How to do it...

We will create a simple function which will launch the factorial execution for the number 200,000, and then continually check if the notification window, with the warning, is up and running via the method of `WaitWindow`.

1. The function we have will look as follows:

```
function testFactorial()
{
    var num = "200000";
    var timeout = 50000;

    var pCalc = Sys.Process("CalcPlus");
    var wCalc = pCalc.Window("SciCalc", "Calculator Plus");

    wCalc.Activate();
    wCalc.Keys("[Esc]");
    wCalc.Keys(num);
    wCalc.Window("Button", "n!").Click();

    while(pCalc.WaitWindow("#32770", "*", -1,
        timeout).Exists)
    {
        Log.Message("Warning message appeared, continuing...");
        pCalc.Window("#32770", "*").Window("Button",
            "*Continue").Click();
    }

    Log.Message(wCalc.Window("Edit").wText);
}
```

2. In the result of the given function execution, the window with the warning will show up several consecutive times (the number depends on how fast the computer is), and at the end, the log will have the result appended.

## How it works...

In the beginning of the function, we initialize the variable of timeout with the value of 50 seconds—namely the time-span needed to wait for the window to appear. If, in 50 seconds, the window with the warning has not shown up, the script assumes the calculation is over, and obtains the results from the text field.



After the entry of the number and clicking on the button for factorial calculations (**n!**), we get into an endless loop waiting for the warning window to appear. After that we call the `WaitWindow` method for the parent object (in our case, it is the object of the process).

The `WaitWindow` method accepts four parameters: the name of the window class, the heading of the window, the position of the window in relation to the other windows of the same class, and the waiting time. In the result of this function execution, the method returns an object with the property `Exists` which we must check in order to find out if the window has appeared.



The `Wait` methods always return an object, not a Boolean value. To check if the window exists, it is necessary to use the `Exists` property, as in our example, or any other suitable property.

If the window is existent (the property `Exists` is equal to `true`), we enter a corresponding message to the log and click on the button **Continue** to go on with the loop execution.

Unlike the `aqUtils.Delay` method, the `Wait` methods are very flexible and allow more efficient usage of the waiting time. For example, if next time around, we have to calculate the factorial value for the number being 10 times greater, we would not have to make any changes in the function (although, the time of execution may exponentially increase, because the complexity of the factorial calculation goes up in geometrical progression).

### There's more...

In our example, we are working with an ordinary Win32 application and use the `WaitWindow` method. For the other types of applications, there exist corresponding `Wait` methods. For example, `WaitWinFormsObject` for .NET applications or `WaitQtObject` for Qt applications. Do always use the correct `Wait` methods, depending on the type of application and the controls element!

`WaitWindow` exists immutably in a singular instance, that is, it always accepts one and the same number of parameters. However, some of the `Wait` methods have several implementations. For example, `WaitQtObject` has three differing implementations with a various set of parameters, each being usable depending on the situation. Before putting the `Wait` methods to use, read up on them in the reference literature in the TestComplete system.

### See also

- ▶ To learn more about waiting objects refer to the *Waiting for a property value* recipe

## Waiting for a property value

Sometimes, an application may be instrumental in a prolonged routine, during which the test should await its completion. Usually, an application prompts users upon completing the work. One of the examples of the completed operations state is a change of the property of the existing controls element (for example, the button `Continue` becomes enabled, background color is changed, and so on).

In this recipe, we will deal with a convenient know-how concerning waiting for the properties to be changed with the help of the `WaitProperty` method.

### Getting ready

Launch Calculator Plus (`C:\Program Files\Microsoft Calculator Plus\CalcPlus.exe`) and make sure it is started in either Standard or in Scientific mode. Make sure that the **Digit grouping** option from the **View** tab is unchecked.

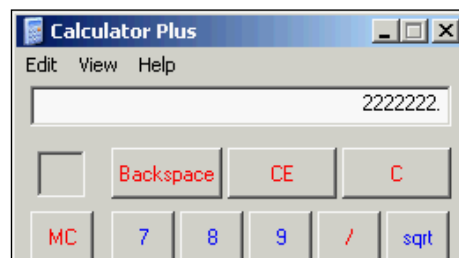
### How to do it...

Our task is inputting one and the same digit into the calculator (for example, the digit 2) until the value in the text output field is equal to **2222222**.

1. To make this happen, we will turn the following script to use:

```
function testWaitProperty()
{
    var expected = "2222222. ";
    var wCalc = Sys.Process("CalcPlus").Window("SciCalc",
"Calculator Plus");
    var edit = wCalc.Window("Edit");

    wCalc.Activate();
    while(!edit.WaitProperty("wText", expected, 1000))
    {
        wCalc.Keys("2");
    }
}
```



2. When the value in the text field becomes equal to the expected one, the function will stop working.

### How it works...

Expected value contains an extra space character at the end of the string - this happens because the calculator applications add this space so that result wouldn't be too close to the right border of the text field. Users don't notice it, but `TestComplete` certainly does.

The method `WaitProperty` takes three parameters:

- ▶ `PropertyName`: This parameter specifies the name of the property whose value is verified
- ▶ `PropertyValue`: This parameter specifies the expected value
- ▶ `Timeout`: This parameter specifies the maximal waiting time

If the property has gotten the expected value, the `WaitProperty` method returns `true`, otherwise, upon expiry of a set time frame, the method returns `false`.

This method is recommended for use instead of an ordinary delay (`aqUtils.Delay`) in the case that the completion of the monitored operation can be tracked by a given property of any of the controls element.

## Mapping custom control classes to standard ones

At times, in tested applications, there crop up controls elements that are unknown in `TestComplete` and therefore, are unworkable; although, externally, a controls element of the kind appears to be a standard one (for example, a text field or a list of elements). It happens because programmers create controls with custom names and `TestComplete` doesn't know how to work with them.

The simplest way to resolve this issue is to try to explain this to `TestComplete`: this class is in reality a standard one (to be more exact, has been inherited from its standard class). In such a situation, we would be working with such an element as if it were a standard one, meaning the one taken after by inheritance. Certainly, in this case, we cannot use the extended possibilities of the controls element in view, while at least, we could execute some standard actions with it. For this purpose, `TestComplete` has specific project settings, which are called **Object Mapping**.

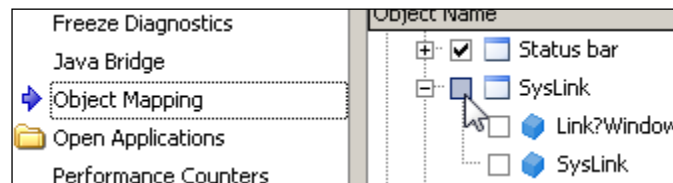
Since creation of a similar controls element or search for existing application to a similar effect is quite a difficult task indeed, we are using a workaround: teaching `TestComplete` not to work with the standard controls elements in order to obtain the data from a controls element of an entirely different type.

In Calculator Plus application (as in the standard Windows calculator), the controls element of the `SysLink` class is being used. This element is visible if the menu item **About** is selected from the **Help** tab. The text **this product is licensed under...** is the element we are talking about. If one were to look up its properties in **Object Browser**, the `wText` property would stand out as containing the text and the parameters, `Link` and `LinkCount`, allowing for more information about the links within the element. This element is our "guinea pig".

## Getting ready

First we need to remove the existing object mapping for the `SysLink` elements as follows:

1. Open the properties of the project (right-click on the name of the project and navigate to **Edit | Properties**).
2. On the toolbar to the right, go to the section **Object Mapping**.
3. Unfold the controls element **Win32 Controls and Windows** in the tree to the right, and uncheck the flag from the controls element of **SysLink**, as shown in the following screenshot:



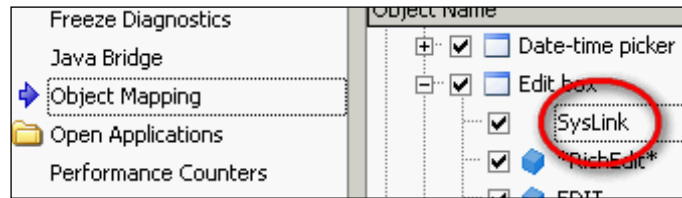
4. Now TestComplete has no idea about recognizing the `SysLink` controls elements.

## How to do it...

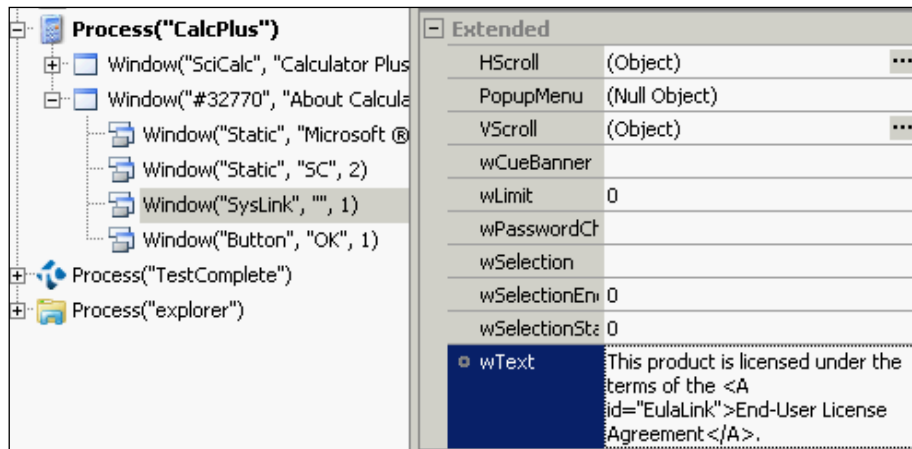
In order to be able to recognize the `SysLink` elements as text fields we need to perform the following steps:

1. Launch Calculator Plus and opt for the menu item **About** in the **Help** tab. The window **About Calculator Plus** will show up on the screen.
2. Open up **Object Browser** and locate the controls element of the `SysLink` type inside the **About** window (this element is unique therein).
3. Look up the property of this element on the panel to the right and make sure that none of them contain the text **This product is licensed under...**
4. Open up project properties (right-click on the name of the project and navigate to **Edit | Properties**).
5. On the panel to the right, go to the **Object Mapping** section.
6. Open up the **Win32 Controls and Windows** controls element, and in it check the element **Edit box**.

- Click on the button **Add Class Name** and into the appearing new controls element input the name of the `SysLink` class, as shown in the following screenshot:



- Again, open up **Object Browser**, make the right-click on the **Sys** element and opt for the **Refresh All** menu item.
- Locate the element of the **SysLink** type again in the tree of objects, and go through its properties. Now we have the `wText` property available, from which we could get a hold on the readable text, as shown in the following screenshot:



### How it works...

Usage of the Object Mapping is the simplest method to work with nonstandard controls elements in case they are inherited from the standard ones.

Usually, in the inherited controls elements, some standard methods and properties still remain, which can be resorted to in order to obtain all (or almost all) the necessary information. This is exactly what we have done in the example of the `SysLink` element: we have taught `TestComplete` to work with this element as with a text field.

In truth we are bereft of some of the possibilities that the former properties held (for example, in the previous-mentioned example, we are short of accessing the `Link` and `LinkCount` properties); nonetheless, quite often all of what's within reach is enough for creation of genuine test scripts.

## There's more...

If Mapping has not been a real help, and you are still up against an unwieldy controls element, you will have to come up with some more advanced methodology, such as:

- ▶ Writing an extension of your own (quite a complicated approach, usually requires help of the developers of the tested application)
- ▶ Usage of the provide properties and methods of the controls element to make it workable
- ▶ Usage of text recognition possibilities and optical recognition
- ▶ Working with the element as with an image

## See also

For more advanced ways of working with nonstandard controls refer the following recipes:

- ▶ The *Working with nonstandard controls* recipe in *Chapter 3, Scripting*
- ▶ The *Using text recognition to access text from nonstandard controls* and *Using Optical Character Recognition (OCR)* recipes

## Using text recognition to access text from nonstandard controls

In some cases, if a controls element in the tested application has not been completely recognizable (for example, nonstandard toolbar), TestComplete can still provide us with access to some of the private properties of the element in view using the **text recognition** method.

In this recipe, we will consider usage of the text recognition method for accessing buttons on the toolbars in Paint application.

## Getting ready

Launch a standard Paint Windows application (navigate to **Start | All programs | Accessories | Paint**).

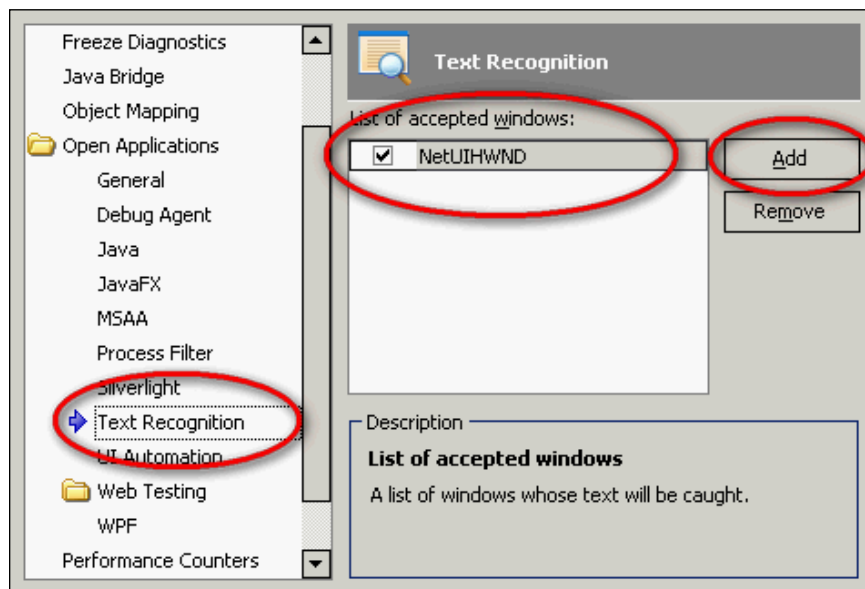
## How to do it...

In order to recognize text from controls we need to perform the following steps:

1. Open up **Object Browser** and click through to the following element therein:

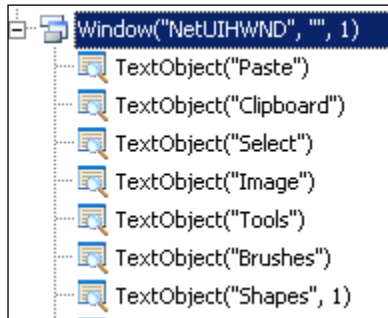
```
Sys.Process("mspaint").Window("MSPaintApp", "Untitled - Paint", 1).Window("UIRibbonCommandBarDock", "UIRibbonDockTop", 3).Window("UIRibbonCommandBar", "Ribbon", 1).Window("UIRibbonWorkPane", "Ribbon", 1).Window("NUIPane", "", 1).Window("NetUIHWND", "", 1).
```

2. The **Window("NetUIHWND", "", 1)** is the panel where the buttons are located; however, we have no possibility to work with these buttons, as the toolbar in view is not a standard one.
3. Now, open the **Project Workspace** tab and pass on to project properties (right-click on the name of the project and navigate to **Edit | Properties**).
4. Open up the group of options **Open Applications – Text Recognition**.
5. Click on the **Add** button and add the class of `NetUIHWND` to the list of **List of accepted windows**.
6. Include this class with the help of the checkbox near the name of the class and save the changes (press `Ctrl + S`) as shown in the following screenshot:



7. Again, open up the **Object Browser** and update the element of **Window("NetUIHWND", "", 1)**, by right-clicking on it and opting for the **Refresh All** menu item.

8. In the result, we will see several sibling objects that now are recognizable by TestComplete, as shown in the following screenshot:



### How it works...

If a nonstandard controls element uses a standard Windows API for text output, we could recognize the text within it with the help of Text Recognition method, as described.

With the elements of the `TextObject` type we can make ordinary actions (for example, triggering mouse-clicks on them, obtaining values of the properties, and so on); however, having no possibility to normally work with their parent objects. For example, in case of the **Select** button in Paint, we are getting access only to the text `Select`, and not to all the buttons with image. Nonetheless, these simple possibilities are usually sufficient for functional testing.

### See also

- ▶ Text Recognition is one of the simplest methods to work with nonstandard controls elements. If this approach could not help you cope the issue of elements recognition, it's time to turn to more sophisticated recipes. For this refer to the *Using Optical Character Recognition (OCR)* and *Dealing with self-drawn controls not supported by TestComplete* recipes.

## Using Optical Character Recognition (OCR)

Optical text recognition (OCR) is usable in cases where other methods of obtaining text from the targeted controls element are not up to the task for whatever reason.

We will supply an example of searching for the **sqrt** text in the calculator window, and making a mouse-click in the center of the located text.



## Getting ready

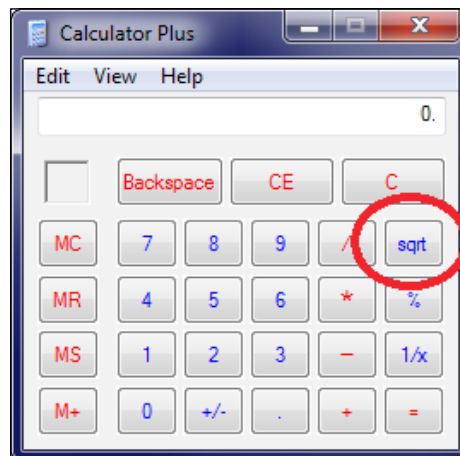
Before embarking on the optical recognition, it is necessary to disable screen font's smooth view which is done as follows:

1. Right-click on the **Computer** icon (on the desktop or in the Start menu) and go for the `Properties` menu item.
2. Click on **Advanced system setting** to the right of the window pane.
3. Go to the **Advanced** tab and click on the **Settings** button in the **Performance** group of options.
4. On the **Visual effects** tab disable option **Smooth edges of screen fonts** and click on **OK**. Besides, it is recommended one uses standard Windows theme instead of classic. To shuffle this off:
  - ❑ Make the right-hand mouse-click on the desktop and opt for the **Personalize** menu item.
  - ❑ Opt for **Windows Classic** in the **Basic and High Contrast Themes** group.
5. Launch Calculator Plus and set the standard mode operandi to it (navigate to **View | Standard**).

## How to do it...

In order to recognize text from a control we need to perform the following steps:

1. First of all, let's enable the calculator window and get the number 16 inputted to it to have a possibility to check that the **sqrt** button has been truly pushed as shown in the following screenshot:



```
var wCalc = Sys.Process("CalcPlus").Window("SciCalc",
    "Calculator Plus");
wCalc.Activate();
wCalc.Keys("16");
```

2. Then we will create a new object OCR for all the calculator window and enter all of the text into the log, that is, the one TestComplete can recognize text in:

```
var calcOCR = OCR.CreateObject(wCalc);
Log.Message("All text from Calc", calcOCR.GetText());
```

Usage of the `GetText` method in this case is not mandatory; however, having all the recognized text in the log can simplify analysis of recognition errors and their underlying causes.

3. Further, we will locate the **sqrt** text, place the found image into the log, and make a mouse-click on it.

```
if(calcOCR.FindRectByText("sqrt"))
{
    var sqrtPic = wCalc.Picture(calcOCR.FoundLeft,
        calcOCR.FoundTop, calcOCR.FoundWidth,
        calcOCR.FoundHeight);

    Log.Picture(sqrtPic, "Found sqrt image");
    wCalc.Click(calcOCR.FoundX, calcOCR.FoundY);
}
```

4. Now, if one were to launch the function, the **sqrt** button would be clicked on and the square root calculations for 16, that is, the result being 4, would be inputted into the results field.

### How it works...

The `FindRectByText` method places coordinates of the located text (`FoundX`, `FoundTop`, `FoundWidth`, and so on) into the object of the `OCRObject` type. Thanks to these coordinates, we have the possibility to make the mouse-click or obtain the image with the targeted text.

It is recommended that one should always place the located image(s) into the log (as we have done so with the help of the `Log.Picture` method); because there always exists a possibility of an incorrect recognition, the image up your sleeve might help you dig out the reason for any incorrect script's behavior, making the task much simpler.

TestComplete by default is capable of recognizing all the symbols of the Latin alphabet, digits, and some specific symbols, covering practically all the standard fonts, sizes, and drawings; however, the quality of recognition may vary from one case to another. Besides, text recognition within images is a prolonged operation.

This is why it is recommended one should apply OCR approach only in some rare and extreme use cases.

## There's more...

If the needed text is recognized in part, there are two methods to resolve the issue of partially incorrect recognition. They are as follows:

- ▶ Usage of the wildcard when passing parameter to the `FindRectByText` method. For example:  

```
calcOCR.FindRectByText("s*rt")
```
- ▶ Usage of an imprecise search and impermissible mistakes. To this end, it is necessary to set the property of `OCROptions.ExactSearch` to be equal to `false`, and in the property of `OCROptions.SearchAccuracy` the value of the error margin should range from 1 (successful match) to 0. For example, the following code would be searching for `sort` text in the calculator window; however, it would click on the button **sqrt**, because text matching would fall within the arranged error margin:

```
var OCROpt = calcOCR.CreateOptions();
OCROpt.ExactSearch = false;
OCROpt.SearchAccuracy = 0.8;
if(calcOCR.FindRectByText("sort", OCROpt))
{
    var sqrtPic = wCalc.Picture(calcOCR.FoundLeft,
        calcOCR.FoundTop, calcOCR.FoundWidth,
        calcOCR.FoundHeight);
    Log.Picture(sqrtPic, "Found sqrt image");
    wCalc.Click(calcOCR.FoundX, calcOCR.FoundY);
}
```

## Dealing with self-drawn controls not supported by TestComplete

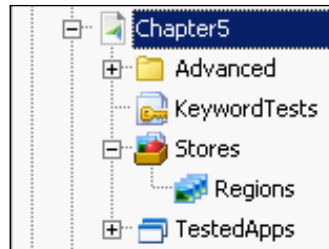
If you have encountered a controls element which is not supported by TestComplete, and has no text to bind to (for example, with the use of text recognition or optical character recognition), working with such an element is possible as with an image, while looking up for other nested images, saved previously. This is done without a tie-up to the parent container-element coordinates, rather operating with the images themselves.

In this recipe we will suppose that no button inside the calculator window is accessible (which means we cannot work with them by using the Window method). First, we would save all the images of the buttons with the numbers to a special element of *Stores*, and then we would undertake searching for these images inside the calculator window image.

### Getting ready

First of all, we need to go ahead and add the new project elements, namely, *Stores* and *Regions*.

1. Right-click on the name of the project and navigate to **Add | New Item**.
2. In the opened window, go to the **Stores** option.
3. Right-click on the added element of **Stores** and navigate to **Add | New Item**.
4. In the opened window, go to the **Regions** option as shown in the following screenshot:



5. Launch Calculator Plus.

## How to do it...

In order to deal with nonstandard controls we will need to perform the following steps:

1. First of all, with the help of a simple function, we will add all of the buttons at once, the images will be worked on later:

```
function prepareOwnDrawnButtonsImages()
{
    var wCalc = Sys.Process("CalcPlus").Window("SciCalc",
"Calculator Plus");
    for(var i = 0; i <= 9; i++)
    {
        Regions.AddPicture(wCalc.Window("Button", i), i.toString());
    }
}
```

Here we have simply gone through all the sibling objects with the corresponding headings, and thus have added their images to `Regions`. In real project we would not have had such a possibility, and this is why we would have to call the `Picture` method for the whole of the calculator window, passing the coordinates of each button as follows:

```
Regions.AddPicture(wCalc.Picture(150, 50, 30, 25),
    i.toString());
```

In the given instance, we have simplified our task for the sake of demonstrating the search for the images.

2. Now, as we have got the stored images of all the necessary controls elements at our disposal, we can search for them inside the calculator window image. To this end, we will need to resort to the `Regions.FindRegion` method. The following function demonstrates how one should click on the button 0, using only the saved image thereof:

```
function testOwnDrawnControls()
{
    var wCalc = Sys.Process("CalcPlus").Window("SciCalc",
"Calculator Plus");
    wCalc.Activate();

    var btnRect = Regions.FindRegion("0", wCalc);
    var xCenter = btnRect.Left + btnRect.Width/2;
    var yCenter = btnRect.Top + btnRect.Height/2;
    wCalc.Click(xCenter, yCenter);
}
```

3. Now we can wrap the code for clicking on the button into a separate function in such a way that we call just one function, without complicating the tests, as follows:

```
function clickWindowButton(window, imageName)
{
    var btnRect = Regions.FindRegion(imageName, window);
    var xCenter = btnRect.Left + btnRect.Width/2;
    var yCenter = btnRect.Top + btnRect.Height/2;
    window.Click(xCenter, yCenter);
}
```

4. In the result, the code for clicking on the calculator button will look as follows:

```
clickWindowButton(wCalc, "5");
```

### How it works...

The `Regions.FindRegion` method allows searching for one image (the first parameter, which is the named element of the `Regions` group, in our case) inside the other.

This method returns the `Rect` object, which is the information repository for the located image (coordinates and the size) in relation to their image, within which the search has been performed.

If the image has not been found, the `Regions.FindRegion` method returns the value `null`.

Since a mouse-click is made in the center of the object, we obtain the coordinates of the located image with the help of the image coordinates and its dimensions.

The advantage of this approach's usability over just clicks on the coordinates of an element is the fact that this approach will be correctly workable even if the size and location of the elements are being changed (you can verify this by changing the calculator mode from Standard to Scientific or vice versa).

### There's more...

We add the images of the buttons to the `Regions` repository with the help of the `Regions.AddPicture` method. This is optionally done via scripts, but can be done manually just as well.

To this end, we need to make a right-hand mouse-click on the `Regions` element, navigate to **Add | New Item**, and then follow the instructions of the wizard.

This is exactly the way we would add the images in case the objects inside the main calculator window would be past recognition.



# 6

## Logging Capabilities

In this chapter we will cover the following recipes:

- ▶ Posting messages to the log
- ▶ Posting screenshots to the log
- ▶ Creating folders in the log
- ▶ Changing log messages' appearance
- ▶ Getting the number of errors in the log
- ▶ Changing pictures' format
- ▶ Comparing screenshots with dynamic content
- ▶ Decreasing log size
- ▶ Generating log in our own format
- ▶ Exporting log to MHT format
- ▶ Sending logs via e-mail

### Introduction

One of the most important feature in every automation tool is logging capabilities. Well-implemented test logs allow easy discovery, help to fix script problems, and identify reasons for failure.

In this chapter, we will discuss frequently used log features and issues one may face while working with TestComplete.



## Posting messages to the log

TestComplete allows committing various types of messages to the log: ordinary messages, warnings, logs, and so on.

In this recipe, we will consider examples of how to use these messages.

### Getting ready

Create a file with the name `myfile.txt` in the root directory of `C:`.

### How to do it...

In order to see examples of all the message types in the log, the following steps should be performed:

1. Create and launch the following function:

```
function testMessages()
{
    Log.Event("An event", "Event additional Info");
    Log.Message("A message", "Message additional Info");
    Log.Warning("A warning", "Warning additional Info");
    Log.Error("An error", "Error additional Info");
    Log.File("C:\\somefile.txt", "A file posted to the log");
    Log.Link("C:\\somefile.txt", "A link to a file");
    Log.Link("http://smartbear.com/", "HTTP link");
    Log.Link("ftp://smartbear.com/", "FTP link");
}
```

In the result, we will get the following screenshot of the log:

Type	Message	Ti... △	Priority	H...	Link
	An event	8:21:50	Normal		
	A message	8:21:50	Normal		
	A warning	8:21:50	Normal		
	An error	8:21:51	Normal		
	A file posted to the log	8:21:51	Normal		<a href="#">File1.txt</a>
	A link to a file	8:21:51	Normal		<a href="#">C:\myfile.txt</a>
	HTTP link	8:21:51	Normal		<a href="http://smartbear.com/">http://smartbear.co</a>

Picture **Additional Info** Call Stack Performance Counters

Error additional Info

## How it works...

In the given example, we have used four different types of messages. They are as follows:

- ▶ **Log.Event:** This message is an event which occurs when TestComplete interacts with a tested application. Usually, messages of this type are placed into the log at the point of text input or mouse-clicks; however, we can also place custom-made events into the log.
- ▶ **Log.Message:** This message is an ordinary message that is usually used for prompting a user concerning current actions that are being executed by the script (usually, of a higher level than that of the events; for example, creation of a user, searching for a record, and so on).
- ▶ **Log.Warning:** This message is a non-critical error. It is used in case the results of the check are different from those expected; nonetheless, execution of the script can carry on.
- ▶ **Log.Error:** This message is a critical error usually used when an error is a critical one, making any further execution of the test would be futile

These four types of message are based on several parameters. The first of them is a string that we observe in the log itself; the second one contains additional information which can be seen in the Additional Info tab, if the message has been clicked on. The second parameter is optional and can be omitted as well as all other parameters.

There are two more types of messages:

- ▶ **Log.File:** This message copies the assigned file into the file with the log, and places a reference-pointer to it. Meanwhile, TestComplete renames the file to avoid naming conflicts, leaving only the original extension intact.
- ▶ **Log.Link:** This message places a link to the web page or a file, without making a copy of the file itself in the folder with the log. On clicking on the link, the file will open with the help of the associated program or a link in the browser.

These two types of message accept the link as the first parameter, and then the message parameters, and those pertaining to the additional information (as the previous four). Only the first parameter is mandatory.

## See also

- ▶ The *Posting screenshots to the log* and *Creating folders in the log* recipes
- ▶ Also, if you want to learn more about additional parameters of different log messages, you can refer to the corresponding help pages by navigating to <http://support.smartbear.com/viewarticle/32871/>

## Posting screenshots to the log

Sometimes, it is necessary to place an image into the log; often, it may be a window screenshot, an image of a controls element, or even that of the whole of the screen. To this end, we use the `Log.Picture` method.

In this recipe we will consider different ways to place an image into the log.

### How to do it...

The following steps should be performed to place an image to the log:

1. First of all, we will create two image objects for the enabled window and the whole of the screen:

```
var picWindow = Sys.Desktop.ActiveWindow().Picture();  
var picDesktop = Sys.Desktop.Picture();
```

2. The image of the active window, now being stored in the `picWindow` variable, will be placed into the log, unchanged:

```
Log.Picture(picWindow, "Active window");
```

3. The image of the desktop is reduced by four times via the `Stretch` method, and then saved on to the file with the help of the `SaveToFile` method:

```
picDesktop.Stretch(picDesktop.Size.Width/2,  
    picDesktop.Size.Height/2);  
picDesktop.SaveToFile("c:\\desktop.png");
```

4. Now we go about creating a new variable of the `Picture` type, loading up an image into it from the earlier saved file, and then placing the same into the log:

```
var pic = Utils.Picture;  
pic.LoadFromFile("c:\\desktop.png");  
Log.Picture(pic, "Resized Desktop");
```

5. As a result of function's execution, the log will contain the two images placed therein: that of the enabled window at the moment of test execution, and that of the reduced desktop copy.

### How it works...

The `Log.Picture` method has one mandatory parameter that is, the image itself; the other parameters being optional.

Images of any of the onscreen objects (of a window, of a singular controls element, of the desktop) can be obtained via the `Picture` method. In our example, with the help of the method, we get the image of the desktop and that of the active window. Instead of the active window, we could use any variable that corresponds to a window or a controls element.

Any image can be saved onto the disk with the help of the `SaveToFile` method. The format of the saved image is determined by its extension (in our case, it is the `PNG`).

If it's necessary to obtain a variable containing the image from the file, we are supposed to create an empty variable placeholder with the help of the `Utils.Picture` property, and then with the help of the `LoadFromFile` method, we upload the image into it. In the future, one could handle the image as any other, received with the help of the `Picture` method.

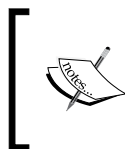
Great-size images can be minified with the help of the `Stretch` method. The `Stretch` method uses two parameters: the new width and height of the image. With the help of the `Size.Width` and `Size.Height` properties, we could zoom in or out on the image in relation to its original size, without setting the dimensions explicitly.

### There's more...

With the help of the `Picture` method, we could obtain not only the image of the whole window or a controls element, but just a part of it. For example, the following code gets an image of the upper left square of the desktop within the sizing of 50 x 50 pixels:

```
var picDesktop = Sys.Desktop.Picture(0,0, 50, 50);
```

The values of the parameters are as follows: coordinates of the left and right top corner, and its width and height.



There is one important project setting which allows automatic posting images in case of error. To enable this option, right-click on the project name, navigate to **Edit | Properties**, click on **Playback** item from the list of options, and enable checkbox **Post image on error**.

Apart from changing the dimensions of the image, `TestComplete` allows for the execution of several, quite complicated imaging manipulations. For example, the comparison of the two images (the `Compare` method), searching for one image inside the other (the `Find` method), and so on. Click on the following link to get to know more about these possibilities:

<http://support.smartbear.com/viewarticle/32131/>

## Creating folders in the log

In some cases, it is useful to conceal some of the messages in the log in such a way that they are not seen all the time, rather than have them be easily accessible. For example, in case of looping through the lines of the table to locate the needed one, we could place each indexed line into the log; however, they could be rather a lot in number, making it unnecessary to view them on the top-level all the time.

To resolve this issue, TestComplete extends a possibility to create special folders in the log, where the messages could be successfully placed.

### How to do it...

The following steps should be performed to create folders in the log:

1. For creation of the new folder in the log, the method of `Log.CreateFolder` is to be used, the folder name expected to be passed as a parameter:

```
var folder = Log.CreateFolder("1st folder");
```






2. Now, for the sake of appropriating all the messages into the folder, it is necessary to enable the same with the help of the method of `Log.PushLogFolder`:

```
Log.PushLogFolder(folder);
Log.Message("This message will appear in the 1st folder");
Log.Event("This event will appear in the 1st folder");
```

3. To close the current folder and revert to the previous one, the method of `Log.PopLogFolder` is to be utilized:

```
Log.PopLogFolder();
Log.Message("This message will appear in the root folder");
```

In the result, the log will assume the shape and form as shown in the following screenshot:

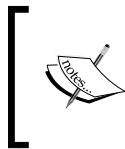
Type	Message
	 1st folder
	 This message will appear in the 1st folder
	 This event will appear in the 1st folder
	 This message will appear in the root folder

## How it works...

The method of `Log.CreateFolder` returns an integer (`Folder ID`), which uniquely identifies the folder. With the help of the `Log.PushLogFolder` method, we enable the folder, whose `Folder ID` has been signified as a parameter.

The folder can be nested inside each other as many times as necessary, thus segregating the information in the log by levels (low-level operations will be the most nested, while the operations of the upper level will be nested above).

Such an approach is really handy, for example, in the case that each test provides a number of steps. Each step will be located on the upper level, while the details of the operations will be hidden out of view. Using such an approach, if an error has emerged, it is easy to see all of it at once: what stage did the error spring up on, since (in case of error generation in the nested folder) the sibling elements are also displayed with an error sign, or with a warning sign, depending on the type of error generated.



Remember to maintain consistency between `PushLogFolder` and `PopLogFolder`, otherwise your results will be pushed to a wrong folder and it will be difficult to find the necessary information in the log!

## There's more...

If you need to send a one-off message into the closed folder, it is not necessary to open it and then close it. All the message methods of the log accept an optional parameter of `Folder ID`, with the help of which one could place messages into the assigned folder. For example, if we had a folder created, as in the previous example, we can place a message into it in the following manner:

```
Log.Message("Posting message to the specified log folder", null,
           pmNormal, null, null, folder);
```

Here, we pass only the message priority (`pmNormal`) and the folder identifier (`folder`), all the optional parameters are set to `null`, which implies that default values will be used.

## Changing log messages' appearance

If you are unhappy with the standard font style and color in the log, they can be changed with the help of the log's attributes. In this recipe, we will deal with outputting the message with changed parameters for the font.

## How to do it...

To post a message with a different style perform the following steps:

1. Create a new object with the help of the `Log.CreateNewAttributes` method and change the following parameters: `Bold`, `FontColor`, and `BackColor` in the following manner:

```
var attrBoldBlue = Log.CreateNewAttributes();  
attrBoldBlue.Bold = true;  
attrBoldBlue.FontColor = clWhite;  
attrBoldBlue.BackColor = clBlue;
```

2. Now we will evoke the method of `Log.Message` and, with the fourth parameter, we will pass the created variable of `attrBoldBlue` to it:

```
Log.Message("Customized message", null, pmNormal,  
            attrBoldBlue);
```

In the result, the message will get outputted into the log as follows:

Type	Message
...	<b>Customized message</b>

## How it works...

With the help of the attributes, we can change the color of the fonts and the background (the properties `FontColor` and `BackColor`) and the text decoration (`Bold`, `Italic`, `Underline`, and `StrikeOut`).

In `TestComplete`, there is a set of supported standard font colors (such as `clWhite` and `clBlue`), which can be used as settings for the font and background color. Similarly, it is possible to change the text display style for other types of messages (errors, warnings, and so on) also.

## See also

- ▶ The complete list of the colors can be found at <http://support.smartbear.com/viewarticle/33785/>

## Assessing the number of errors in the log

TestComplete allows you to retrieve the number of errors for the current test item (with the help of the `Log.ErrCount` property); however, there is no way of finding out the total number of errors in all the executed tests. Such a possibility can be useful only if a certain predefined number of errors is treated as critical, upon reaching which test execution should be stopped altogether.

### How to do it...

First of all, we will add two new variables on the level of Project Suite as follows:

1. Right-click on the name of the Project Suite (in the **Project Workspace** toolbar, to the left) and navigate to **Edit | Variables**.
2. On the **Temporary Variables** list, right-click on the **New Item** menu item.
3. In the **Name** field, input the name of the `ErrorsTotal` variable, into the field **Type** select `Integer`, and set the **Default Value** field to 0.

This variable will be a counter of the errors.

4. Similarly, add the variable with the name of `ErrorsMax`. As **Default Value**, set the number of errors which should signal stopping test execution (in our example, those are equal to 3).

In the result, we will have two new variables created, as shown in the following screenshot:

Name	Type	Default Value	Description
ErrorsTotal	Integer	0	Errors counter
ErrorsMax	Integer	3	Maximum number of errors allowed

5. Now we will create a handler for the `OnLogError` event, which will increase the counter of the errors. For this purpose, perform the following steps:
  1. Add the `Events` element to the project, if it is still missing (right-click on the name of the project and navigate to **Add | New Item | Events**).
  2. Navigate to **Events | General Events** on the element. In the result, the events panel will be opened.
  3. In the right section of the panel (the **Events to Handle** column), unfold the element of **General Events** and highlight the event of `OnLogError`.
  4. Click on the button **New** inside the `OnLogError` element.



5. In the opened window **New Event Handler**, select the module in which you will store the event handler, and then click on the **OK** button.
6. In the result, we will have an empty function created with the name of `GeneralEvents_OnLogError`.
7. Change the function in the following manner:

```
function GeneralEvents_OnLogError(Sender, LogParams)
{
    ProjectSuite.Variables.ErrorsTotal += 1;
    if(ProjectSuite.Variables.ErrorsTotal >
        ProjectSuite.Variables.ErrorsMax)
    {
        Runner.Stop();
    }
}
```

8. Now we will write a simple function which will create 10 error messages in the log:

```
function testErrorsCount()
{
    for(var i = 0; i < 10; i++)
    {
        Log.Error("Error #" + i);
    }
}
```

9. If we launch the `testErrorsCount` function now we would see that upon the fourth emergent error, test execution would be stopped, since the number of arisen errors exceeded the preset value of the `ErrorsMax` variable.

### How it works...

In the `OnLogError` events handler, we increase the number of emerging errors each time the error is generated in the log (no matter how: either with the help of the method `Log.Error` or via `TestComplete` in itself, the event will be processed all the time).

When the number of errors exceed the preset threshold, we stop execution of the tests with the help of the `Runner.Stop` method.

To the variables on the level of Project Suite there exists access from any project. Therefore it is unimportant how exactly tests are launched: from the Project Suite, or just a project or several separate functions. In any case, the variable will be updated each time an error occurs.

It's worthwhile to note, if you have several projects running in Project Suite, each one of them should contain some sort of handler. To this end, it is much easier to create such a handler in one project and in the other projects add an existing module (by right-clicking on the `Script` element, thus opting for the **Existing Item** menu item in the **Add** menu). This allows us to avoid code duplication.

### There's more...

In the same manner, one could trace the number of other messages (Event, Warning, File, and so on), since for each of them there exists a corresponding property, containing their number in the current test item (`Log.EvnCount`, `Log.WrnCount`, `Log.FileCount`, and so on).

### See also

- ▶ To know more about event handlers in TestComplete, refer to the *Creating even handlers* recipe in *Chapter 12, Events Handling*

## Changing pictures' format

In TestComplete, there are several components and processing images such as testing log, region checkpoints, and Visualizer. In all the cases, TestComplete uses a specific image format for storing pictures.

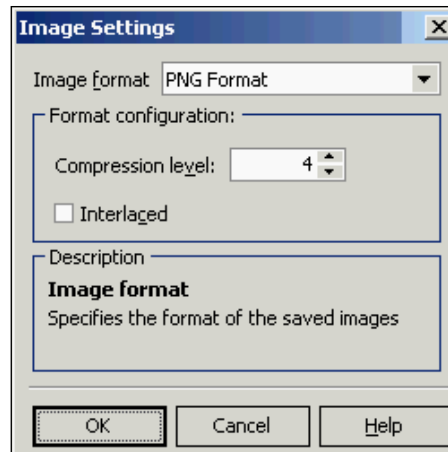
In this recipe we will learn the ropes concerning the alteration of these images' format, which TestComplete is processing.

### How to do it...

In order to change pictures' format we need to perform the following steps:

1. Navigate to **Tools | Options** and open the group of **Engines** in the **General** options. Here in the **Images** group, the currently used format is displayed (for example, PNG).
2. Click on the **Configure** button in the **Images** group.

3. In the opened **Image Settings** window, opt for the necessary format.
4. In the group of **Format Configuration** options, set other parameters for the selected image (for example, quality or depth of the color gamut).



5. Click on **OK**.
6. Now TestComplete will use the currently selected format.

## How it works...

TestComplete supports four image formats, each of them having certain advantages and shortcomings as follows:

- ▶ **BMP:** This is the uncompressed format, used in cases where we need precision in imaging. The major drawback of this format is its humongous size.
- ▶ **JPEG:** This is the most thrifty format (from the viewpoint of the size of the file); however, its image can be less precise.
- ▶ **PNG:** This is a better trade-off in-between quality and size criteria and is usually used when you need to have images with lossless compression.
- ▶ **TIFF:** This is used in specific cases (for example, for typography), if it's necessary for the project in a certain case.

For the vast majority of cases, the variant with the PNG format is of a greater avail than others.

## There's more...

If, in majority of the cases, you are better off with one of the economizing formats (PNG or JPEG), and it is seldom you need to create precise images in the BMP format, you can still change the format of the images on the fly.

The following example demonstrates creation of a screenshot with the use of the following two differing formats:

```
function testPictureFormat()
{
    Options.Images.ImageFormat = "BMP";
    Log.Picture(Sys.Desktop.Picture(), "BMP screenshot");
    Options.Images.ImageFormat = "PNG";
    Log.Picture(Sys.Desktop.Picture(), "PNG screenshot");
}
```

## Comparing screenshots with dynamic content

Let's suppose you need to compare windows images or those of controls elements with some dynamic contents (that is, if part of the image is being changed every time). This could be the case when a given controls element displays current date or time. In this situation, the image would be different every time; nonetheless, there is a way to resolve this holdup.

## Getting ready

First we will do some preparations:

1. Launch Calculator Plus in the standard mode (navigate to **View | Standard**).
2. Add the element of **Stores** (right-click on the name of the project and navigate to **Add | New Item | Stores**).
3. Add the element of **Regions** to your project (right-click on **Stores** and navigate to **Add | New Item | Regions**).
4. Launch the following code (it will create an image of the main calculator window in **Regions**):

```
var wCalc = Sys.Process("CalcPlus").Window("SciCalc");
Regions.AddPicture(wCalc.Picture(), "Calculator");
```

5. Enter any digit to the calculator (different from the data inputted initially at the moment of image creation) and launch the following code:

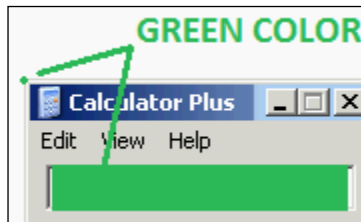
```
var wCalc = Sys.Process("CalcPlus").Window("SciCalc");  
Regions.Compare("Calculator", wCalc.Picture());
```

6. In the result, we will get the error **The regions are not equal.**

### How to do it...

Now we are going to prepare the image so that we can compare it by performing the following steps:

1. Open the Calculator file in any graphic editor (for example, Paint), which was created via the method `Regions.AddPicture` (it is located in the folder with the project, subfolder to `Regions`).
2. Select any color which is not used in the window (for example, green).
3. Fill out the upper-left pixel of the image with this color.
4. With the same color, fill out all the dynamic area (in our case, the whole of the text field).



5. Save the image and launch the following script:

```
var wCalc = Sys.Process("CalcPlus").Window("SciCalc");  
Regions.Compare("Calculator", wCalc.Picture(), true);
```

6. In the result, in the log, we will see the message **The regions are the same.**

### How it works...

If the `Transparent` parameter (the third parameter of the `Regions.Compare` method) would be set to `true`, `TestComplete` reads the color of the upper-left pixel of the image and interprets the same as `transparent`, that is, does not include the areas of the color in the check.

As a result, the images are considered the same, although a part of them is different.

## There's more...

Sometimes, it is not sufficient to add transparent areas, for example, if the dynamically changing data appears in different parts of the screen.

To resolve the task, the `Regions.Compare` method has another parameter, namely that of `Tolerance`. This should be an integer number, which is the maximal number of the different pixels. In this case, TestComplete will assume that images are the same, unless the number of the differing pixels exceeds the value of the `Tolerance` parameter.

## Decreasing log size

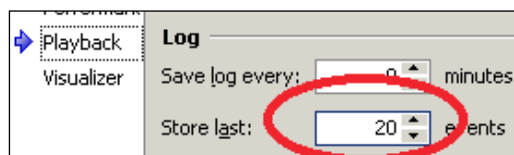
When having a great deal of tests, the size of the log, as generated by TestComplete, can be really big.

In this recipe we will take into consideration several methods to decrease the size of the log.

## How to do it...

In order to decrease the log file size perform the following steps:

1. **Disable the Visualizer:** To this end, open the properties of the project (right-click on the name of the project and navigate to **Edit | Properties**), click on the **Visualizer** group of parameters, and set the **Collect Test Visualizer data during test run** option to the value **Off**.
2. **Disable the events generation:** To do so, open the properties of the project (right-click on the project and navigate to **Edit | Properties**), click on the **Playback** group of parameters, and set the option of **Store last** to a value that is different from zero (for example, 20).



3. **Change the format of the used images:** To this end, open the properties for the project (right-click on the name of the project and navigate to **Edit | Properties**), click on the group of **General** parameters, click on the **Configure** button, and in the **Image format** field, select the format, different from that of BMP. If this is insufficient, you could diminish the quality of the stored images (parameters of **Compression level** and **Compression quality**, depending on the format).

## How it works...

Each of the mentioned actions influences the project in a different manner:

- ▶ **Visualizer:** This action allows you to automatically save objects' screenshots with which TestComplete is interacting; however, in the majority of cases, this is quite redundant. It suffices to get a screenshot of the main screen in case of an error. To this end, in the project's properties, in the **Playback** section, it is necessary to enable the option of **Post image on error**.
- ▶ **Events:** They are generated quite frequently (upon each interaction with the tested application), which means this also takes up some space. By setting a certain value into the field of **Store last ... events**, we get the possibility to review the assigned number of the last events in case of an emerging error; all the other events will be erased.
- ▶ **Image formats:** They also influence the size of the logs. For example, the BMP format is not recommended to be used in the majority of cases, since images of this format, take up lots of space on the HDD. The most economizing format is that of JPEG.

## Generating log in our own format

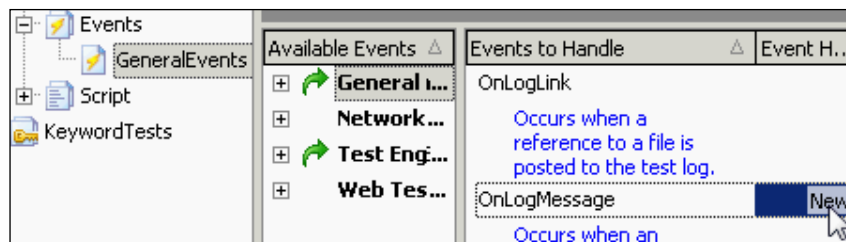
In this recipe, we will undertake the study of a simple example of generating our own log in the text format, which could be used in addition to the standard TestComplete log or even instead thereof.

For this purpose, we need to redefine the standard events.

## How to do it...

In order to generate our own log we need to perform the following steps:

1. Add the element **Events** to your project, if it has not been added yet (right-click on the name of the project and navigate to **Add | New Item | Events**).
2. Double-click on **Events** in the **General events** element.
3. In the right part of the window, unfold the element of **General events** and select the **OnLogMessage** element.



4. Click on the button **New** inside the **OnLogMessage** element.
5. In the opened **New Event Handler** window, select the module in which the new handler will be located, and click on the **OK** button.
6. In the result, within the selected module, there appears the following function:  
GeneralEvents\_OnLogMessage

7. Write the following code for the function:

```
function GeneralEvents_OnLogMessage (Sender, LogParams)
{
    var logFilePath = "c:\\tclog.txt";
    if(!aqFile.Exists(logFilePath))
    {
        aqFile.Create(logFilePath);
    }
    var logFile = aqFile.OpenTextFile(logFilePath,
        aqFile.faReadWrite, aqFile.ctUTF8);
    logFile.WriteLine(aqDateTime.Now() + " MESSAGE: " +
        LogParams.Str);
    logFile.Close();
}
```

8. Similarly, create other handlers for all the types of messages which you would like to have outputted into the external log-file (OnLogError, OnLogEvent, OnLogWarning, and so on).
9. Write and launch a simple function, which will call the next methods Log.Message, Log.Warning, and all the other redefined handlers. For example:

```
function testOwnLog()
{
    Log.Message("Demo message");
    Log.Warning("Demo warning");
}
```

As a result, in addition to the ordinarily generated log, we will have the following file created: C:\tclog.txt, where each log message will be duplicated in the file.

### How it works...

First, we check if the file of the log exists, and then create it, if needed. Hereafter, we open it in "read-and-write" mode.

Into this file, we place the current date and time and the text of the message (which we obtain from the method of LogParams.Str), after which we go ahead and close the file.



This is quite an inefficient method of working with the file, since for each of the messages we have to open and close it, which is telling on the performance; however, this example demonstrates the possibility of creating our own log with the help of intercepting logging events.

In a true-to-life project, we would need to think through a more advanced variant. For example, storing the messages into an array and saving the changes onto the disk, if the array reaches a certain amount of messages.

### There's more...

If you would like to block output of the information into a standard TestComplete log on top of duplicating the messages into a separate file, it is enough to write the following line in the beginning of the handler:

```
LogParams.Locked = true;
```

## Exporting log to MHT format

If you need to make a report, generated by TestComplete, to somebody who has no TestComplete pre-installed, it would be more convenient to generate it in a format which can be portably opened on any computer.

TestComplete allows saving reports in the MHT format (archived HTML), which can be opened in any version of Internet Explorer browser.

### How to do it...

In order to generate an MHT file we need to perform the following steps:

1. Create and launch the following function:

```
function testExportResults()  
{  
    Log.Message("Message 1");  
    Log.Error("Error 1");  
    Log.SaveResultsAs("c:\\results.mht", lsMHT);  
}
```

In the result, we get the log in TestComplete window and the file `c:\results.mht` with similar contents.

2. Open the following file `c:\results.mht` (by double-clicking on its name in the Explorer window) and make sure it can be opened in the browser (Internet Explorer, by default).

## How it works...

The method of `Log.SaveResultsAs` stores the log into the file of the MHT format, regardless of the number of scripts launched; therefore this method should be evoked just once at the end of working of all the scripts.

As the first parameter, the full name of the file has to be signified, the second parameter assigns the following format: `1sMHT`, `1sHTML`, or `1sXML`. The most convenient format is that of MHT, as a result, just one file will be generated, which is viewable on any computer.

If there arises an error during the export, the `Log.SaveResultsAs` method returns `false`.

## Sending logs via e-mail

The simplest method to send an e-mail in TestComplete is that of `BuiltIn.SendMail`; however, it is appropriate only in case the mail server requires no authentication.

This is why in the given recipe we will consider a universal method to dispatch logs with the help of **Collaboration Data Objects (CDO)**. The example we use will employ the Gmail mail server.

## How to do it...

We need to perform the following steps to send an e-mail:

1. To begin with, we will write the function which sends the mail. To this end, we will need to correctly fill out all the fields of the CDO object. Configuration is to the following effect:

```
function SendEmail(mFrom, mTo, mSubject, mBody, mAttach)
{
    var schema, mConfig, mMessage;
    schema = "http://schemas.microsoft.com/
        cdo/configuration/";
    mConfig = Sys.OleObject("CDO.Configuration");
    mConfig.Fields.Item(schema + "sendusing") = 2;
    mConfig.Fields.Item(schema + "smtpserver") =
        "smtp.googlemail.com";
    mConfig.Fields.Item(schema + "smtpserverport") = 465;
    mConfig.Fields.Item(schema + "smtpauthenticate") = 1;
    mConfig.Fields.Item(schema + "smtpusessl") = true;
    mConfig.Fields.Item(schema + "sendusername") =
        "MYGMAIL@gmail.com";
}
```

```
mConfig.Fields.Item(schema + "sendpassword") =
    "MY_PASSWORD";
mConfig.Fields.Update();

mMessage = Sys.OleObject("CDO.Message");
mMessage.Configuration = mConfig;
mMessage.From = mFrom;
mMessage.To = mTo;
mMessage.Subject = mSubject;
mMessage.HTMLBody = mBody;

aqString.ListSeparator = ",";
for(var i = 0; i < aqString.GetListLength(mAttach); i++)
    mMessage.AddAttachment(aqString.GetListItem(mAttach, i));
mMessage.Send();
}
```

2. Now we will archive the current log with the help of the `slPacker` object:

```
slPacker.PackCurrentTest("c:\\testresults.zip");
```

3. Hence, we send the resulting archive with the help of the earlier written `SendEmail` function:

```
SendEmail("MYGMAIL@gmail.com", "destination@example.com",
    "Test results", "Email body", ["c:\\testresults.zip"]);
```

In the result, from the address of `MYGMAIL@gmail.com` to the address of `destination@example.com`, a letter will be sent with the archive attached, containing the tests results.

### How it works...

In our case, the `gmail.com` server was used as an example, since it is one of the most frequently used mail services. However, the parameters of each of the servers vary, and this is why one should consult with the system administrator of your network (if you send letters from your corporate mail system).

The filled out object of `CDO.Configuration` is passed as a `Configuration` property to the object of `CDO.Message`. This same object has parameters of the letter preassigned (the address of the sender and the recipient, the theme and the body of the letter, and the files attached), after which the method `Send` is to be called to `CDO.Message`.

The method `slPacker.PackCurrentTest` allows archiving the results of the current test (this is necessary for more convenient manipulations of the attachments, as handling a single file is easier than several).

If all the parameters are signified correctly, the e-mail will be sent.

Naturally, instead of packing up logs, it is also possible to export them into the MHT file and send it via e-mail.

### See also

- ▶ To learn how to export logs into MHT format, read the *Exporting log to MHT format* recipe



# 7

## Debugging Scripts

In this chapter we will cover the following recipes:

- ▶ Enabling and disabling debugging
- ▶ Using breakpoints to pause script execution
- ▶ Viewing variables' values
- ▶ Debugging tests step by step
- ▶ Evaluating expressions

### Introduction

Debugging is one of the most important actions when you create or modify test scripts. It allows you to run tests step by step, view variables' values, evaluate complex expressions related to variable and objects' states, and so on.

In this chapter we will discuss the most frequently used techniques and approaches of debugging available in TestComplete. These techniques are similar for most development environments.



If you use TestComplete 8 or earlier, you need to install Microsoft Script Debugger to enable debugging in TestComplete. You can download Script Debugger from the following link: <http://www.microsoft.com/en-us/download/details.aspx?id=22185>.

## Enabling and disabling debugging

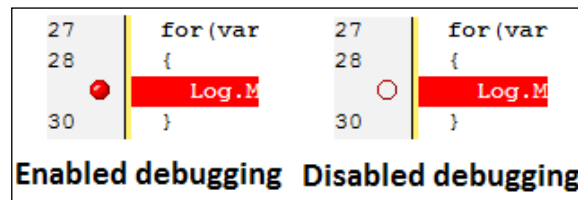
Sometimes in the process of script debugging you might add a lot of breakpoints, which then intervene in running the test without pauses.

If you want to have the breakpoints disabled, but do not want to disable all breakpoints one-by-one, it is enough to disable the debugging mode.

### How to do it...

Enabling and disabling the debugging mode in TestComplete is quite simple and is done as follows:

1. To deactivate the debugging mode, navigate to **Debug | Enable Debugging**. Meanwhile, all the breakpoint icons will change (instead of the solid circle, an empty circle will appear).
2. To switch back to the debugging mode, navigate to **Debug | Enable Debugging** again. In between, the icons of the breakpoints will again become solidly painted.



### How it works...

As the debugging mode is rendered inactive, TestComplete will ignore all the breakpoints, allowing test execution entirely without pausing and without removing the breakpoint itself.

### There's more...

If you are launching automated tests to be executed from the command line, it is usually unacceptable to have any stops during script run. To avoid such stops, it's enough to pass the `/silentMode` parameter to TestComplete. In this case, breakpoints are also ignored.

## Using breakpoints to pause script execution

To pause automated test execution at a specific point, the breakpoints are applied. In this recipe we will learn to work with the breakpoints in TestComplete.

### Getting ready

Create the following function:

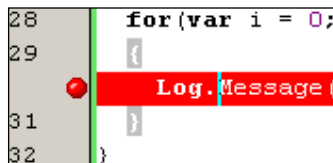
```
function testBreakpoints()
{
    for(var i = 0; i < 10; i++)
    {
        Log.Message("Iteration #" + i);
    }
}
```

This function will simply output the messages 10 consecutive times.

### How to do it...

To demonstrate working with the breakpoints, perform the following steps:

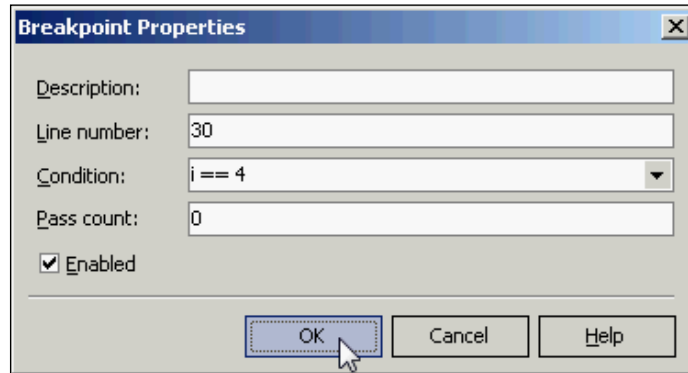
1. Click on the line with the `Log.Message` method call and press the `F9` key (or click on the column with the numbers to the left of the editor, opposite the previous line). In the result, the line will be highlighted in red against its background, and in the column to the left, a red circle will appear, as shown in the following screenshot:



2. Launch the function for execution. The function execution will pause at the first breakpoint in the script.
3. Stop the script execution (for example, by pressing the buttons combination: `Shift + F2`). Make sure the log contains only the error message **Script execution was interrupted.**, since we have stopped the script without ever executing the `Log.Message` line.
4. Right-click on the red circle of the breakpoint and select the **Properties** menu item.



- In the opened **Breakpoint Properties** window, input the value of `i == 4` in the **Condition** field and click on the **OK** button:



- Pay attention that the icon of the breakpoint has changed, and now contains the **f** symbol:



- Launch the function again and break its execution as soon as the script execution has been paused.
- Pay attention that besides the error message in the log, there are several messages with the following text: **Iteration #N**.

## How it works...

In the first instance we have used a simple breakpoint. It fires as soon as script execution reaches the given code line and pauses the execution, pending further action from the user.

In the second instance, we have used a conditional breakpoint, which allows pausing the execution only in case a certain condition is met (in our case, the variable `i` should be equal to 4).

Conditional breakpoints are very convenient in those cases when we have a loop, which is executable in part by pausing the execution under conditions specified.

Besides a condition of the kind, one can make use of another condition of Pass Count. In this field, it is possible to assign the number of loops of the script, after which the breakpoint should fire (for example, in our case, we could simply set the value to 4 in the **Pass Count** field).

## There's more...

There also exists another method for setting a conditional pause into the script execution, namely that of the `Runner.Pause` method. The result of this is analogous to the workings of the breakpoint.

If you have a breakpoint, which is of use from time to time (for example, a non-stable script, often requiring debugging), we can simply disable the breakpoint instead of completely removing it. To this end, it is necessary to right-click on it and select the **Enable** menu item.

## See also

- ▶ The examples of the situations when we need breakpoints are also available in the *Viewing variables' values* and *Debugging tests step by step* recipes

## Viewing variables' values

During the script debugging, there often arises a need to look at the current values of the variables used in the script.

In this recipe we will deal with several methods of doing this.

## Getting ready

Launch a standard Windows Notepad application (`C:\Windows\notepad.exe`).

## How to do it...

In order to familiarize ourselves with the lookup possibility of the variable values, we will need to write a small script with the use of different variable types.

1. Write the following function:

```
function testDebug1()
{
    var pNotepad = Sys.Process("notepad");
    var num = 15.8;
    var str = "string value";
    var bool = true;
    Log.Message("Dummy message");
}
```

2. Set the breakpoint on the last line, where the `Log.Message` method call takes place (set the cursor on the line and press `F9` button).

3. Launch the function. In the result, the function execution will pause on the line with the breakpoint and we will be able to look up the values of all the created variables.
4. In the lower part of the TestComplete window, open the **Watch List** tab.
5. Right-click on the empty list and select the **New Item** menu item.
6. In the **Expression** field of the opened **Watch Properties** window, enter the name of the `pNotepad` variable and click on **OK**. The variable will show up in the list of those being tracked.
7. Similarly, add all the other variables (`num`, `str`, `bool`, and so on).

As a result, all the variables will appear on the list. One can also look up their values and types there (if TestComplete can determine the value for the given variable type).

Expression	Value	Type
<input checked="" type="checkbox"/> num	15.8	
<input checked="" type="checkbox"/> str	string value	
<input checked="" type="checkbox"/> bool	True	
<input checked="" type="checkbox"/> pNotepad	(Object)	Object
ChildCount	3	Integer
CommandLine	"C:\Windows\system32\notepad.exe"	String

### How it works...

To the **Watch List** list we can add any variables and even entire expressions for the lookup of their current values. For example, if we enter `num+str` as an expression, the values of the same will be tantamount to **15.8 string value**, since TestComplete will automatically transmute the number to the string and join two string-type expressions together.

If we are working with the objects (`pNotepad`, in our instance), we can also look up their values via the uncollapsed view of the tree of the sibling objects, as shown in the previous screenshot.

Values on **Watch List** are updated automatically, and this is why usage of the list is especially handy when using loops, when it's required to track the variable value at each iteration.

### There's more...

For quick lookup of the values of a given variable, it is not necessary to add it to **Watch List**, as it's enough to hover the mouse cursor over the name of the variable at any place of the script and TestComplete will show the tool tip with the value of the variable at hand.

Another method to quickly view the values of the variables is the `Locals` listing, in which all the local variables are shown (that is, the variables from the current function). The `Locals` list can be found on the same panel where **Watch List** is located.

## See also

- ▶ The *Evaluating expressions* recipe

## Debugging tests step by step

In this recipe we will consider scripts debugging by doing a set of step-by-step instructions. This is necessary if we are left with some vague results after test execution, and a simple variable values lookup, at a given point of time, is insufficient.

## Getting ready

To begin, we will require two functions, one of which is to be called from the other:

1. Let's create the first function `fn1`:

```
function fn1()
{
    var n = 5, s = "str";
    Log.Message(n);
    Log.Message(s);
    return s;
}
```

2. Let's create the second function `main` so that it calls the first one twice, and then outputs the results into the log, as obtained due to the `fn1` function call:

```
function main()
{
    var res1 = fn1();
    var res2 = fn1();
    Log.Message(res1);
    Log.Message(res2);
}
```

## How to do it...

In order to perform step-by-step execution in the debugging mode, it is necessary to complete the following actions:

1. Set the breakpoint on the first line of the `main` function (to this end, set the cursor at this line, and press `F9` button).
2. Launch the `main` function. In the result, the function execution will pause.

3. Press *F10*. As the result, the line where the cursor is set will be executed, and the `fn1` function will come through; the `res1` variable will have the `str` value assigned.
4. Now the cursor is located on the following line with the second call of the `fn1` function.
5. Press *F11*. In the result, the cursor will relocate into the `fn1` function, and we will have a possibility to execute the function step by step by pressing *F10* key.
6. Press *F5*. This will bring off the continuation of the function execution till the very end.

## How it works...

In the debugging mode, we can perform the following commands:

- ▶ **Step Over** (*F10*): This command executes the entire line of code and allows transition to the following code statement.
- ▶ **Step Into** (*F11*): This command steps into the function that is being called in the current line (if possible). If one cannot step into the function, the `Step Over` command is to be executed.
- ▶ **Continue Execution** (*F5*): This command continues the script execution from the current place down to the next breakpoint.
- ▶ **Run to Cursor** (*F4*): This command goes on executing the function from the current line to the place where the text cursor has been set.

## Evaluating expressions

Most often, to look up the variables' values in the debugging mode, it is enough to resort to the **Watch List** and **Locals** possibilities. For more complex tasks, we can use the **Evaluate** window.

## Getting ready

Create the following function:

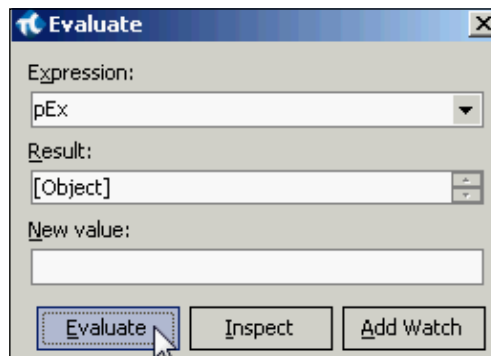
```
function testEvaluate()  
{  
    var pEx = Sys.Process("explorer");  
    Log.Message("");  
}
```

and set the breakpoint on the second line of the function (set the cursor on this line and press *F9*).

## How to do it...

To get acquainted with the possibilities of the **Evaluate** window, we will need to implement the following actions:

1. Launch the created `testEvaluate` function. Its execution will be stopped and `TestComplete` will switch to the debugging mode.
2. Navigate to **Debug | Evaluate** or press the following combination of keys: `Ctrl + F12`. In the result, the **Evaluate** window will appear on the screen.
3. Input `pEx` into the **Expression** field and click on the **Evaluate** button. In the result, the **Result** field will get the current value of the `pEx` variable (`[Object]`) inputted.



4. If we click on the **Inspect** button, the screen would have the **Inspect** window displayed, which is analogous to that of **Object Browser**, where all the properties and methods are viewable.
5. Close the **Inspect** window and input the "string value" string (by all means, with the quotation marks!) into the **New value** field and press *Enter*.

In the result, the **string value** line will show up in the **Result** field. Now, the `pEx` variable contains a new value.

## How it works...

The **Evaluate** window extends wider possibilities of working with various expressions as compared to that of the **Watch List** list:

- ▶ The first major difference consists in the possibility of assigning the variables with the new values, while the variables' types can be altogether different (for example, in our case, the variable of `pEx` first contained the object, and then the line).

- ▶ The second difference consists in the **Inspect** window. In this sense, it resembles **Object Browser**; however, we can use it to look up not only the sibling objects of the `Sys` element, but rather any other objects as well. For example, it is possible to enter the `BuiltIn` value into the **Expression** field and look up all the properties and methods of the `BuiltIn` object, which are extended by `TestComplete`.

Also, from the **Evaluate** window, one can add a current expression to the **Watch List** list (with the help of `Add Watch`) for further tracking.

# 8

## Keyword Tests

In this chapter we will cover the following recipes:

- ▶ Recording and understanding Keyword Tests
- ▶ Adding new actions to existing Keyword Tests
- ▶ Enhancing Keyword Tests using loops
- ▶ Creating object checkpoints
- ▶ Calling script functions from Keyword Tests
- ▶ Converting Keyword Tests to scripts
- ▶ Creating our own Keyword driver

### Introduction

Automated tests are sometimes created by people who do not have experience in programming. For such cases TestComplete has a feature called Keyword Tests.

On the one hand, Keyword Tests are designed to be easily created and supported in a visual way, when users don't need to write code. On the other hand, Keyword Tests have almost the same abilities as Script Tests do: we can record them, modify them, and debug them.

Another advantage of the Keyword Tests is that they look the same for different types of application (Win32, .NET, Web, and so on), thus allowing us to concentrate on the tests themselves.

In this chapter we will consider some of the most important aspects of Keyword Tests and finally will create our own Keyword driver with user-defined keywords.



## Recording and understanding Keyword Tests

**Keyword-driven Testing** is such an approach to automated tests creation that goes together with visual representation of user-actions.

Each action is twofold involving the following two parts:

- ▶ An object, which is being handled
- ▶ An action, which is necessary to accomplish for the object at hand

The simplest way to create a Keyword Test is recording it. In this recipe, we will learn on a simple example of a Keyword Test.

### Getting ready

Launch the Calculator Plus application (C:\Program Files (x86)\Microsoft Calculator Plus\CalcPlus.exe).

### How to do it...

To record the Keyword Test, it is necessary to complete the following actions:

1. Navigate to **Test | Record | Record Keyword Test** menu item. This will kick off the recording.
2. Switch to the **Calculator Plus** window and click on several buttons (for example, **6, +, 3, =**).
3. Stop the recording by clicking on the button **Stop** on the **Recording** panel.
4. Click on the **OK** button in the opened **Create Project Item** window for automatic creation of the **NameMapping** project element.
5. As a result, we will have the new **Keyword Test** created.

Item	Operation	Value	Description
CalcPlus			
wndSciCalc			
btn6	ClickButton		Clicks the 'btn6' button.
btn	ClickButton		Clicks the 'btn' button.
btn3	ClickButton		Clicks the 'btn3' button.
btn2	ClickButton		Clicks the 'btn2' button.

## How it works...

Each operation in Keyword Test is represented in TestComplete by the four columns:

- ▶ **Item:** This is the controls element which will undergo the action. In our example, it is mapped to the following four buttons: **btn**, **btn1**, **btn3**, and **btn6**; which are sibling objects of the **wndSciCalc** window. The **wndSciCalc** window, in its turn, is a sibling object of the **CalcPlus** process.
- ▶ **Operation:** This is the action we perform over the object. In our example, we are performing the action of **ClickButton** throughout.
- ▶ **Value:** This stands for optional parameters, which are passed to the action. For the **ClickButton** action no parameters are implied.
- ▶ **Description:** This is a more detailed explanation of the executed action.

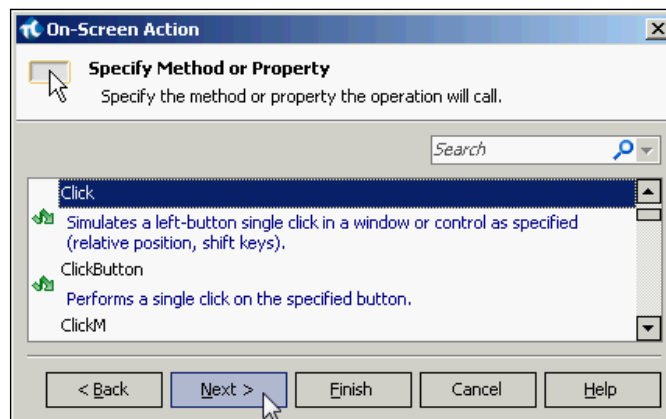
## There's more...

If, in the process of recording, you have committed some wrong actions (for example, by having clicked on a wrong button, as in case with the calculator), there is no need to redo the recording of the test all over again. The objects and actions are easily changeable in TestComplete editor.

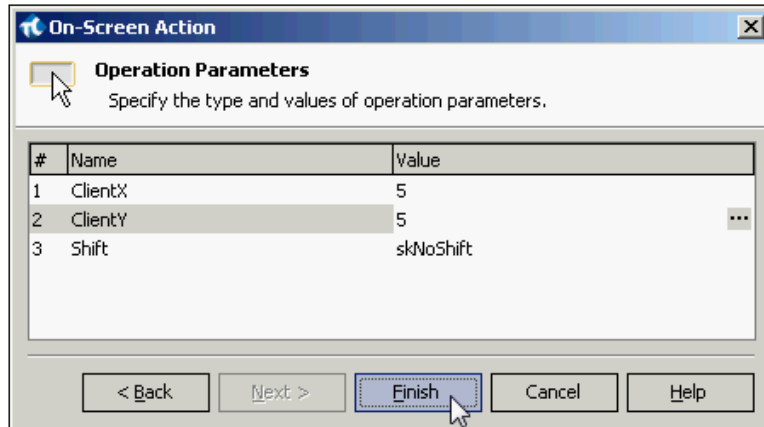
Let's take the **ClickButton** operation for an example. This operation will trigger a mouse-click in the center of the object. If we would like to click in the preassigned button coordinates, we would have to use the **Click** operation, which accepts the parameters for the given button-click coordinates.

To change the used operation for the **btn6** button perform the following steps:

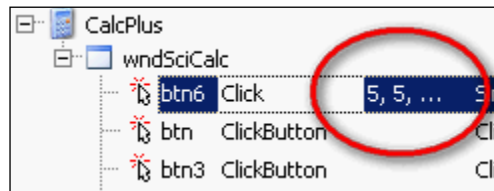
1. Double-click on the name of the operation.
2. In the **On-Screen Action** window, select the **Click** element and click on **Next**, as shown in the following screenshot:



- Assign values for the parameters **ClientX** and **ClientY** and click on **Finish**. These values should not exceed the object's width and height, these can be viewed in the **Object Browser**:



- Now, the click on the **btn6** button will always be made in the coordinate of **5,5**:



The object (item) can be changed the same way.

## Adding new actions to existing Keyword Tests

In this recipe we will deal with the manual addition of actions to the Keyword Tests without resorting to the means of recording scripts. As an example, we will consider an operation of a mouse-click on the **CE** button.

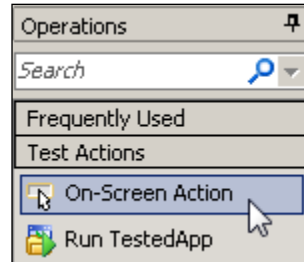
### Getting ready

Create a new empty Keyword Test (right-click on the **Keyword Tests** element, **Add | New Item**) and launch the Calculator Plus application ("C:\Program Files (x86)\Microsoft Calculator Plus\CalcPlus.exe").

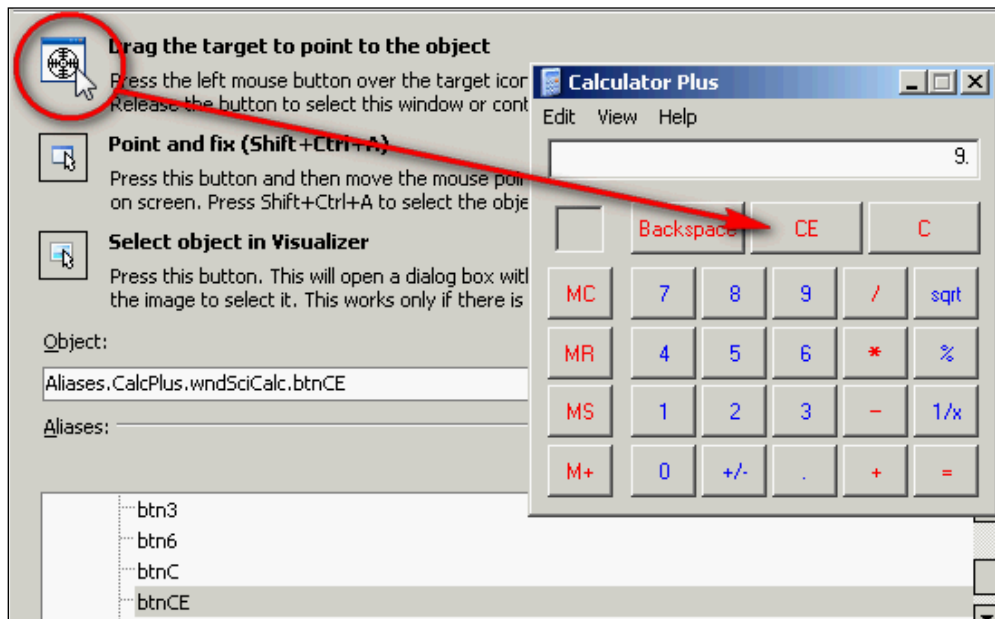
## How to do it...

To add the operation for clicking on the button, it is necessary to perform the following steps:

1. Select the **Test Actions** on the **Operations** toolbar.
2. Double-click on the **On-Screen Action** element:



3. Drag-and-drop the sign of the target onto the button of **CE** and click on **Next**:



4. On the list of available methods, leave the **ClickButton** element selected by default, and click on the **Finish** button.

5. In the result, we will have the created action appearing in our test:

Item	Operation
<ul style="list-style-type: none"> <li>[-] CalcPlus                             <ul style="list-style-type: none"> <li>[-] wndSciCalc                                     <ul style="list-style-type: none"> <li>[+] btnCE ClickButton</li> </ul> </li> </ul> </li> </ul>	

### How it works...

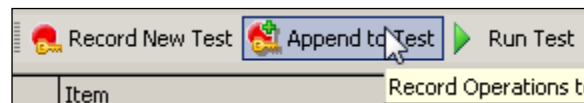
It is not only possible to record the actions in the Keyword Tests through the recording means—they can be created manually as well. Meanwhile, we need to signify a specific object for each action that we are handling: including the action we undertake and their parameters (if they are necessary for the action in view).

In the **Operations** list, there are several available operations:

- ▶ **Test actions:** These are operations used to work with screen elements, tests, objects search, and execution of code snippets
- ▶ **Logging:** These are actions used to work with the log (output of messages, screenshots, and so on)
- ▶ **Web:** This is a group of actions used to work with the browser
- ▶ **Checkpoints :** These are used to create various checks in the tests
- ▶ **Statements:** These are analogues of the `for`, `while`, `if...then...else`, `try...catch` loops, and statements
- ▶ **Miscellaneous:** These are actions for other possibilities (working with the Indicator, insertion of a delay, and so on)

### There's more...

Sometimes, it is easier to add new actions to test by recording them into the existing test, other than manually creating those. To this end, we need to click on the **Append to Test** button on the Keyword Test toolbar and record the actions:



The actions can be easily reordered by simply dragging them to a new location, yet any keyword test can be run from any step by right-clicking on the step and selecting **Run** from the selected operation menu item.

## Enhancing Keyword Tests using loops

Loops are used for the purpose of executing repetitive actions in a test. With the help of the operations related to the **Statements** group, we can add such actions to the Keyword Test.

In this recipe, we will take up an example of clicking on one and the same button for several consecutive times with the help of the `For` Loop.

### Getting ready

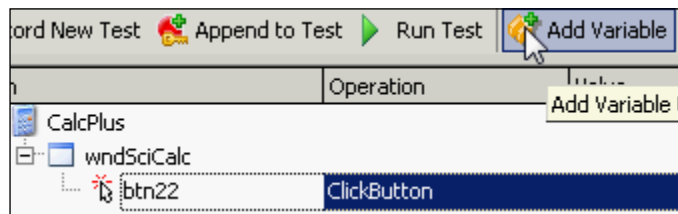
First, let's record a simple script for further modification:

1. Launch the Calculator Plus application: (C:\Program Files (x86)\Microsoft Calculator Plus\CalcPlus.exe).
2. Begin the recording (**Test | Record | Record Keyword Test**).
3. Click on the **2** button in the calculator and stop the recording by clicking on the **Stop** button on the **Recording** toolbar.

### How to do it...

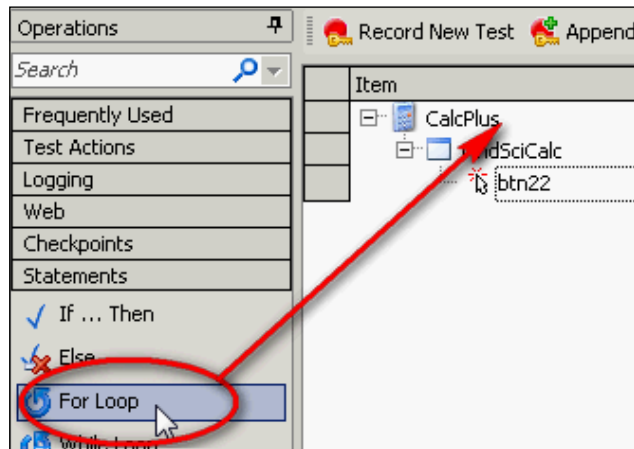
For creation of the `for` loop, it is first necessary to create a counter-variable:

1. Click on the **Add Variable** button on the Keyword Test toolbar:

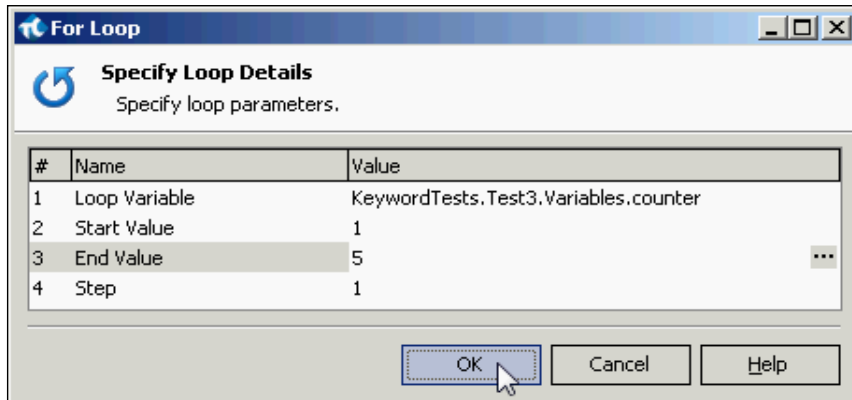


2. In the open **Add Variable to Keyword Test** window, input the name of the variable (`counter`) and the type (`Integer`), and click on **Finish**.
3. Now we will add the loop itself.

- In the list of **Operations**, opt for the **Statements** group and drag-and-drop the **For Loop** element onto the **CalcPlus** element:

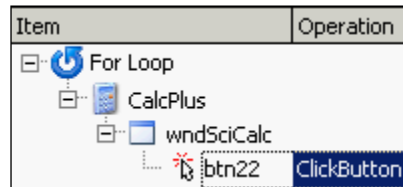


- In the **For Loop** window, input the loop parameters: **Loop Variable** by selecting the previously created variable (`counter`), **Start Value** (1), and **End Value** (5):



- Click on the **OK** button.

7. Now we need to place the **ClickButton** operation inside the loop.
8. Select the **CalcPlus** element and press the following keys combination: *Ctrl* + right arrow key.
9. In the result, the **ClickButton** operation will become a sibling element of the **For Loop**:



10. Now, if you would launch the test to be executed, the **2** button will be clicked down five times in a row.

### How it works...

The **For Loop** in the Keyword Test is workable in the same manner as the `for` loop in any kind of programming. Specific actions, placed inside the loop, are executed for a pre-set amount of times. To assign a specific number of iterations, a loop variable is to be used (in our case, it is the `counter` variable). During each iteration the value of the variable is incremented by the value of **Step** (we have left it to be equal to 1 by default). When the value of the loop variable is greater than the **End Value**, the loop execution is terminated.

In a similar way the **While Loop** and **If...Then** condition can also be created.

## Creating object checkpoints

As in run-of-the-mill scripts in Keyword Tests, we can add the checkpoints. In this recipe, we will consider a simple example of **Object Checkpoint** creation.

### Getting ready

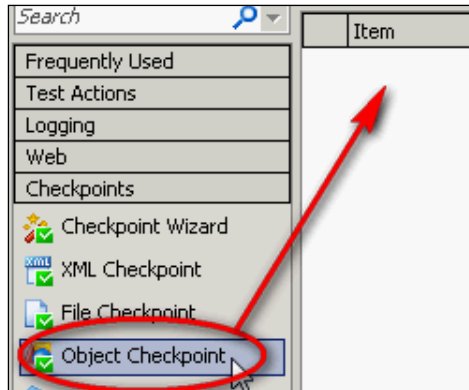
Launch the Calculator Plus application (`C:\Program Files (x86)\Microsoft Calculator Plus\CalcPlus.exe`).



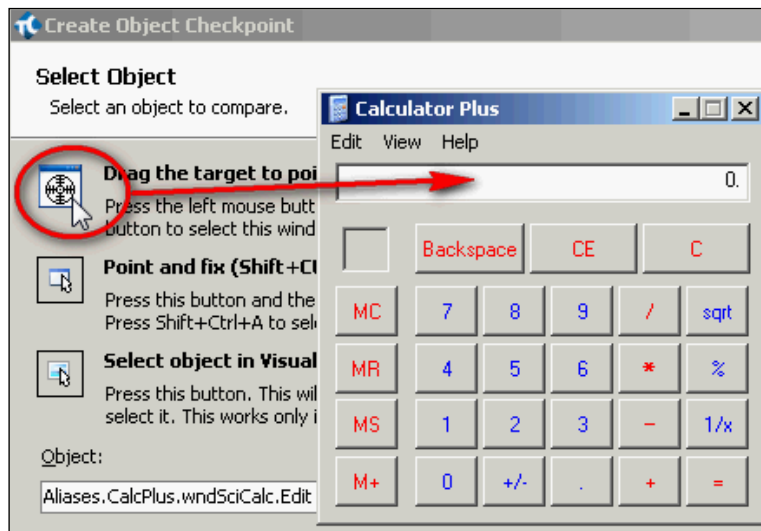
## How to do it...

To create an **Object Checkpoint**, it is necessary to complete the following actions:


1. In the list of **Operations**, select the **Checkpoints** element and drag-and-drop the **Object Checkpoint** element to the place in the test where you are going to add the checkpoint (in our example, we are using a new test with no actions whatsoever).



2. In the opened **Create Object Checkpoint** window, drag-and-drop the sign of the target onto the text field of the calculator.
3. Click on **Next** three times, and then click on **Finish**.



- In the result, checkpoints of the object, that is, the text field of the calculator, will be added to the test.

Item	Operation	Value
 Object Checkpoint	Edit 1	Aliases.CalcPlus.wndSciCalc.Edit

### How it works...

With the help of **Object Checkpoint**, it is possible to check several properties of the object at once (for example, text, visibility, availability, and any others).

The object itself will be stored in the project element of **Stores | Objects**. In the future, all the verifiable properties can be changed, if needed.

### See also

- In this recipe we have taken up the process of **Object Checkpoint** creation, and that—quite briefly. If you would like to learn in greater detail about creation parameters and checkpoint editing possibilities, refer to the *Creating object checkpoints* recipe from *Chapter 3, Scripting*.

## Calling script functions from Keyword Tests

Sometimes, it is easier to write up a simple function with the help of a programming language than implement the same functionality with the help of the keywords in the Keyword Tests.

In this recipe we will consider how to call an ordinary function from the Keyword Test.

### Getting ready

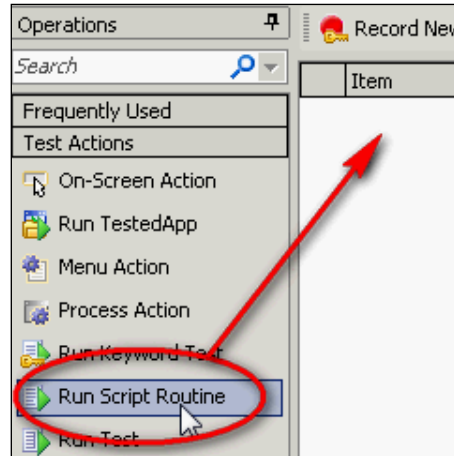
Create the following function in the module of **Unit1**:

```
function testRunFromKeywordTest(param1)
{
    Log.Message("Parameter: " + param1);
}
```

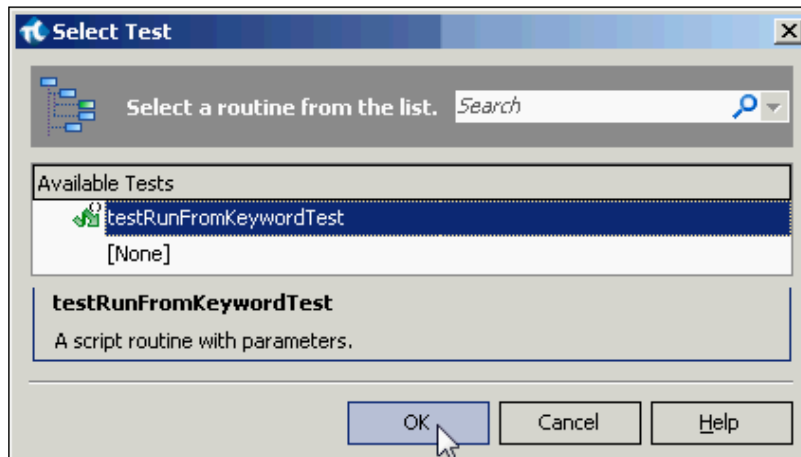
## How to do it...

To evoke the function, it is necessary to do the following actions:

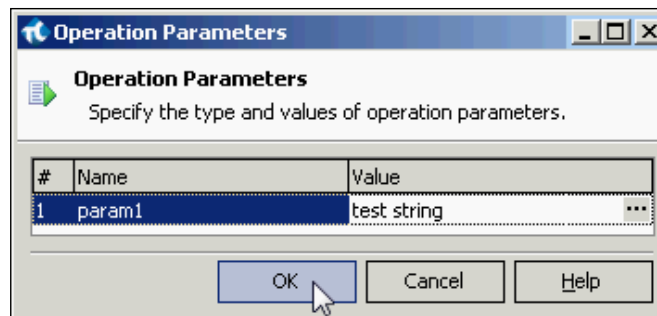
1. In the **Operation** list, select the **Test Actions** element and drag-and-drop the **Run Script Routine** element to the spot, where you plan to call the created function.



2. In the opened **Select Test** window, select the necessary function and click on **OK**:



3. If the function accepts parameters, a new **Operation Parameters** window will open up, where it is necessary to signify the values of the parameters of the called function, and then click on **OK** again:



- In the result, the call of the function will be added to the test.

If the test should be launched, the log would contain the following message: **Parameter: test string.**

### How it works...

With the help of the **Run Script Routine** operation, we can call any function from the current project.

If it's necessary to call the function from a different project, we must add the respective module to the current project by beforehand right-clicking on the **Script** element, and navigating to **Add | Existing Item** menu item.

### There's more...

Besides calling the ordinary functions, Keyword Tests allow us to call code snippets, without creating individual functions (the **Run Code Snippet** operation), as well as the methods of screen objects (the **Call Object Method** operation). It is also possible to call other Keyword Tests by using the **Run Keyword Test** action from the **Test Action** group.

If you want to use values calculated earlier as parameters, you can use Project variables or Script variables to store the value and then specify it by clicking on the button with three dots next to the **Value** field.

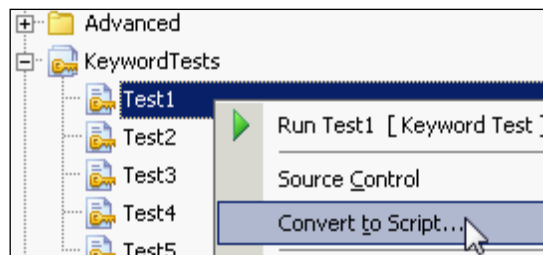
## Converting Keyword Tests to scripts

If you have used the Keyword Tests, and made up your mind to switch to writing scripts, you can easily convert the existing scripts into the scripting format.

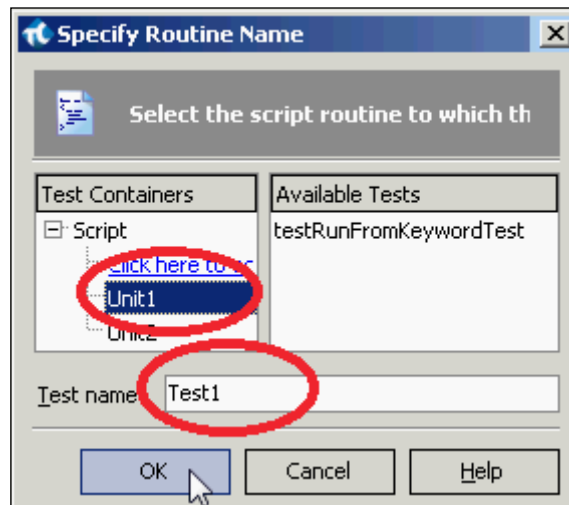
### How to do it...

To convert the Keyword Test to a script:

1. Right-click on the name of the test and select the **Convert to Script** menu item.



2. In the opened **Specify Routine Name** window, select the module into which you would like to save the script, assign it with a name and click on **OK**.



3. In the result, the selected module will contain the created function with all the converted actions.

## How it works...

With the help of the [Click here to add a new script unit](#) link, it is possible to create a new module immediately at the point of converting the test. At the point of converting the Keyword Test to a script, the same language is used, as then pre-selected for the whole project.

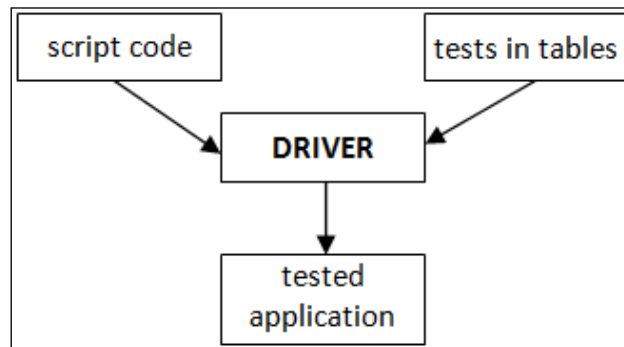


Please keep in mind that the reverse operation (script converting to a Keyword Test) is not possible!

## Creating our own Keyword driver

**Keyword-driven Testing** does not only consist in possibilities that are provided by TestComplete. In a wider sense, Keyword-driven testing is such an approach to testing: when we are creating our own keywords, and in the result, tests creation is nothing short of evoking these keywords. Ideally, tests are created as tables, where each line stands for one test action (call of the keywords with parameters). Apart from the tables, we also have the code written with the help of the programming language and the driver (a class, function or several functions) used, which read keywords from the tables and call the corresponding code from the project.

Schematically, the simplified Keyword-driven approach will appear as follows:



The major advantage of the approach consists in the fact that the code of the scripts and the driver are written and maintained by programmers, while the tests can be created by anyone, including those who are not well-cognizant in programming (that is, testers, product specialists, or customers).

In this recipe we will take up a simple example of creating our own Keyword-driven Tests, for consideration. As a tested application, we will use a standard Notepad application.

## Getting ready

We will create three components for our keyword-driven framework:

- ▶ Functions for launching and closing the Notepad
- ▶ Test in Excel table, where we will store the necessary calls of the keywords
- ▶ Driver-function that will read the steps from the test one-by-one and execute the necessary actions

Also, for our example, it is necessary to create the `C:\somefile.txt` file with any contents.

## How to do it...

First and foremost, we will create a function to work with the Notepad:

1. The function `startNotepad` will launch the Notepad and open the file that was passed as a `fileName` parameter:

```
function startNotepad(fileName)
{
  if(fileName == undefined)
  { fileName = ""; }
  Win32API.WinExec("notepad.exe " + fileName, SW_SHOWNORMAL)
}
```

2. The `closeNotepad` function closes all the open copies of the Notepad:

```
function closeNotepad()
{
  while(Sys.WaitProcess("notepad", 500).Exists)
  { Sys.Process("notepad").Terminate(); }
}
```

3. The `checkNotepadFileName` function checks if the correct file has been opened:

```
function checkNotepadFileName(expectedName)
{
  var np = Sys.Process("notepad").Window("Notepad");
  aqObject.CompareProperty(np.WndCaption, cmpStartsWith,
  expectedName);
}
```

4. Now we will define the keywords. We will need the `OPEN`, `CLOSE`, and `CHECK` keys (as they correspond to the operations, which are to be executed in the test: open the Notepad, check if the correct file has been opened, and then close the Notepad). The keywords `OPEN` and `CHECK` will also have their parameters.

5. From these keywords we will create tests and make them up into Excel table. In the result, our test will appear in the following manner:

	A	B	C
1	Step	Action	Param1
2	1	OPEN	
3	2	CHECK	Untitled
4	3	CLOSE	
5	4	OPEN	c:\somefile.txt
6	5	CHECK	somefile.txt
7	6	CLOSE	

6. The last stage: driver creation. In our case, this is one and the same function, which scampers through all the lines of the assigned table and executes all the instructions one-by-one:

```
function driver(fileName, table)
{
    var steps = DDT.ExcelDriver(fileName, table);
    while(!steps.EOF())
    {
        var stepNum = steps.Value("Step");
        var action = steps.Value("Action");
        var param = steps.Value("Param1");

        switch(action)
        {
            case "OPEN":
                startNotepad(param);
                break;
            case "CHECK":
                checkNotepadFileName(param);
                break;
            case "CLOSE":
                closeNotepad();
                break;
            default:
                Log.Error("Unknown action: " + action);
        }
        steps.Next();
    }
}
```



7. Now, everything is all-set, and we can go ahead and launch the created test. To this end, we will need but a single line:  

```
driver("C:\\notepad.xls", "Sheet1");
```
8. In the result, the Notepad will be launched twice: first, without the parameters (a new document will be created); and second, the Notepad will open up the `C:\\somefile.txt` file.

### How it works...

The driver calls the functions that correspond to each keyword, which is encountered in the test. Fine-tooth-combing of the file is made possible with the help of the DDT driver, provided by TestComplete.

The tests themselves are separated from the code, that is working with the tested application, this is why they are easy to comprehend even for a person who is not learned in programming (quite naturally, the keywords should correspond with the actions that are to be performed at the call of the keyword).

### There's more...

This is a very simple example, wherein we have the basics of the Keyword-driven approach demonstrated. In more complex situations (for example, when testing several or more multiple-components applications), more complicated approaches are usually used. For example, several self-standing drivers, each being meant for a separate application or a component, or a more complex structure of the tables (for example, we can add an **Application** column, wherein the presently workable application is to be signified), and so forth.

### See also

- ▶ Reading up on DDT, and the used driver, in greater details, is possible in the *Chapter 9, Data-driven Testing*

# 9

## Data-driven Testing

In this chapter we will cover the following recipes:

- ▶ Generating random data for tests
- ▶ Accessing a specific cell in a table
- ▶ Reading all data from a table
- ▶ Using DDT tables for storing expected values
- ▶ Changing CSV delimiter and other parameters
- ▶ Driving data without using loops
- ▶ Accessing Excel spreadsheets without having MS Office installed
- ▶ Auto-detecting Excel driver

### Introduction

When performing functional testing of any application, it is often required that we use various input parameters to verify that the application produces correct results in all cases.

TestComplete has a specific feature for storing such data, it is called **Data-driven Testing (DDT)**. In case of using DDT approach, data can be stored in different database-like storages (Excel files, CSV files, or databases). To access data in these files, TestComplete provides a special object DDT.

In this chapter we will consider different tasks related to DDT approach. Most of our examples will use Excel for storing data, but these principles can be easily extended to other types of storage.

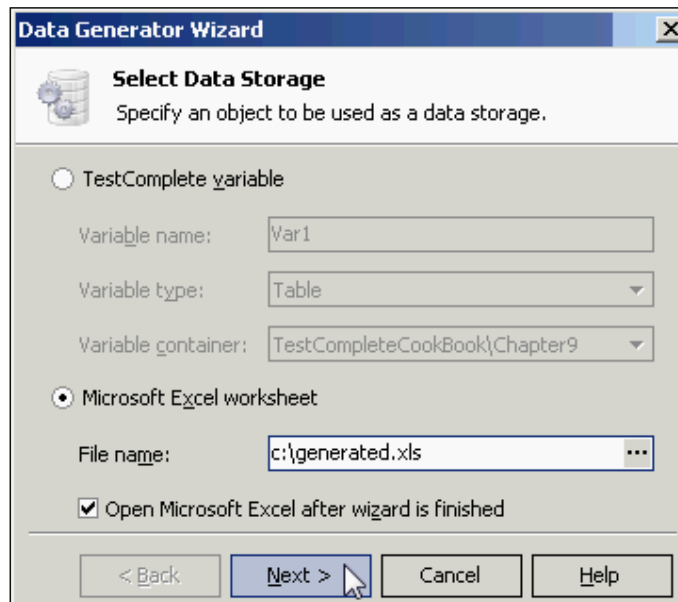
## Generating random data for tests

If you need to generate a great deal of various data, TestComplete will come in mighty handy with a special tool called **Data Generator**.

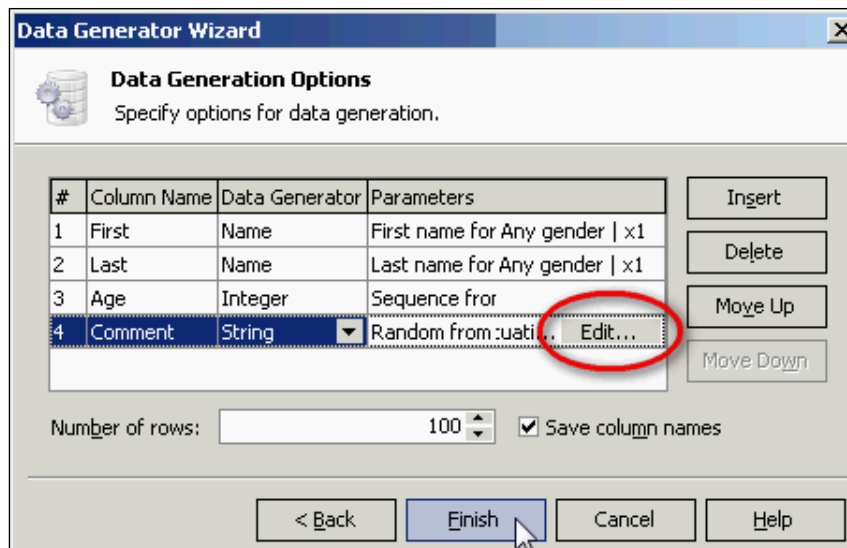
### How to do it...

Let's suppose we need to generate a list of 100 people. To this end:

1. Select the **Test | Generate Data** menu item.
2. In the opened **Data Generator Wizard** window, select the **Microsoft Excel worksheet** option, and input the wanted file by name and then click on **Next**:



3. On the following screen, with the help of the **Insert** button, add as many fields as you need, and then rename them (for example, Last Name and First Name).
4. For each of the fields, assign the appropriate data type (*String*, *Integer*, and so on) depending on the data type you are going to store in this field. With the help of the **Edit** button inside each data line, it is possible to customize the parameters of the generated data (intervals for numbers, length and type of the symbols for strings, and so on).
5. In the **Number of rows** field signify the number of lines (100, in our case), that you would like to have generated, then click on **Finish**:



6. In the result, a file with all the necessary data will be generated for us:

	A	B	C	D
1	First	Last	Age	Comment
2	Valentine	Mckinney	1	Pk\z'\$tE=4BG#L&bM
3	Keane	Burns	2	SxSx\$b_dnax,)
4	Benedict	Whitley	3	sMR@
5	Vernon	Berry	4	fb`5YG]q*C-yBiS^#@]

### How it works...

TestComplete allows generation of different data types (strings, names, numbers, addresses, cities, and so on), diversifying their parameters within a wide range (the interval for the numbers, types of the symbols for strings, and so on).

Besides, it is possible to save the selected parameters with the help of the **Save to User-defined** button in order to have an available template for new data generation at your disposal in the future.

## Accessing a specific cell in a table

In this recipe, we will consider creation of a function, which allows reading values out from a single cell in an Excel file with the help of the DDT capabilities.

### Getting ready

Create an Excel file `c:\readcell.xls` with the following contents:

	A	B	C	D
1	First	Last	Age	Position
2	John	Doe	36	Manager
3	Norma	Noe	30	Developer
4	Jack	Smith	32	Developer
5	Carla	Coe	25	Tester

### How to do it...

In order to retrieve a value from a specific cell:

1. It is necessary to first of all, get your bearings straight with function declaration. We will need to specify the name of the file, name of the spreadsheet, the number of the line and that of the column:

```
function readCell(file, sheet, row, col)
```

2. The body of the function will be to the following effect:

```
var data = DDT.ExcelDriver(file, sheet);
var currentRow = 1;
var value;
while(!data.EOF())
{
    if(currentRow == row)
    {
        value = data.Value(col - 1);
        DDT.CloseDriver(data.Name);
        return value;
    }
    currentRow++;
    data.Next();
}
Log.Error("Row #" + row + " not found");
DDT.CloseDriver(data.Name);
```

3. Retrieving the value from the cell in the script will look as follows:

```
Log.Message(readCell("C:\\readcell.xls", "Sheet1", 4, 4));
```

This example will get the word **Tester** outputted to the log (the contents of the D5 cell).

### How it works...

The only method to retrieve the value from the targeted line via the DDT is looping through all the previous lines. To this end, we make use of the `while` loop and the `currentRow` variable (the number of the current line has to be calculated independently, as `TestComplete` does not extend such a possibility).

The `Value` property allows retrieval of the value from an assigned column (by referring to its heading or a column number).

As soon as we reach the targeted line `if(currentRow == row)`, we store the value from the current line and the specified column and then quit the loop.

Pay attention to the use of the `Next` and `CloseDriver` methods:

- ▶ The `Next` method places the file pointer to the next line. If, at the end of the loop, we forget to evoke this method, the loop will carry on endlessly in the first line of the data file.
- ▶ The `CloseDriver` method closes the driver, averting overflow above the maximally allowed number of the opened drivers (the maximal number thereof can stand at 64) and releases the file to be possible to access by other applications.

Also, pay attention to the fact we have started lines numbering from one (`var currentRow = 1`), thus, we pre-suppose that numbers of the columns also 1-based (`data.Value(col - 1)`): because in `TestComplete` columns are 0-based. We have permuted this behavior, as indexing to start with one seems to be more logical.

### See also

- ▶ If you have to retrieve values from singular cells in an Excel file quite frequently, it is much better to read all of the contents of cells to an array, and then retrieve the necessary data from it (see the following recipe *Reading all data from a table*).

## Reading all data from a table

If it is necessary to frequently read data from files (Excel or CSV), it is better to once read the data to an array, and then address the array to screen its elements for the retrieval.

In this recipe we will take up an example of assigning the data from Excel file to an array in such a manner that the elements of the array are easily accessible via the use of the line number and the column heading of the table.

### Getting ready

Create an Excel file `c:\readall.xls` with the following contents:

	A	B	C	D
1	First	Last	Age	Position
2	John	Doe	36	Manager
3	Norma	Noe	30	Developer
4	Jack	Smith	32	Developer
5	Carla	Coe	25	Tester

### How to do it...

To read the data from the file into an array, we will need to go about the following actions:

1. First open the file with the help of the `DDT.ExcelDriver` method (to this end, we will need the path to the file and the name of the spreadsheet).
2. With the help of the `while` loop, iterate through all of the lines of the file, each time adding a new element to the `lines` array.
3. For each of the elements of the `lines` array that correspond to the line, create the properties with the names that tally up with the names of the columns. For example, if the table has a `FirstName` column, then each element of the array will have the `FirstName` property.
4. In the result, the function will appear as follows:

```
function DDTReadAll(file, sheet)
{
    var lines = [];

    var data = DDT.ExcelDriver(file, sheet);
    while(!data.EOF())
    {
```

```
var line = {};  
for(var i = 0; i < data.ColumnCount; i++)  
{  
    var colName = data.ColumnName(i);  
    line[colName] = data.Value(colName);  
}  
lines.push(line);  
data.Next();  
}  
return lines;  
}
```

5. The following example demonstrates use of the function:

```
var people = DDTReadAll("C:\\readall.xls", "Sheet1");  
Log.Message(people[1].First);
```

### How it works...

This example will output the message **John** to the log (the name from the first line of the file). After having read the data once into the `people` variable, we can address similarly any other fields and lines.

The same approach to handle the data retrieval from a file allows, first of all, to bring up to speed accessing the data since we need to just once open the file and read all of its content out, after which referrals are made inside the computer memory; secondly, this will improve code readability due to the comprehensible names of the properties and lines indices.

Besides, it is much easier and handy to manipulate the data that are being stored in the variables, rather than with the data that is being stored in a file.

## Using DDT tables for storing expected values

Most often, the Data-driven approach is used to store the expected values in the process of testing. In this case, all the testing data are to be stored in one place, which really facilitates their overhaul and modification on as need-be-basis.

In this recipe we will consider an example of storing expected values for computing via the Calculator application.



## Getting ready

Launch the Calculator Plus application in the Standard mode by selecting **View | Standard** and create the file `c:\test_expected_values.xls` with the following content:

	A	B	C
1	id	Expression	Result
2	1	2+7-5	4
3	2	3**4-1	11
4	3	124/2+30	92
5	4	31-20*4	44

## How to do it...

To complete the task, we will need to create the following three functions:

1. The `testExpectedValues` function contains the main logic, it loops through all the lines in the table, reading out the values from the cells:

```
function testExpectedValues()
{
    var data = DDT.ExcelDriver("C:\\test_expected_values.xls",
"Sheet1");
    clickCalcButton("C");
    while(!data.EOF())
    {
        var expression = data.Value("Expression");
        var result = data.Value("Result");

        for(i = 0; i < expression.length; i++)
        {
            var button = expression.substr(i, 1);
            clickCalcButton(button);
        }
        clickCalcButton("=");
        checkCalcResult(result);
        data.Next();
    }
    DDT.CloseDriver(data.Name);
}
```

2. The `clickCalcButton` function simulates clicking on a button in the calculator, which has been passed as a parameter:

```
function clickCalcButton(button)
{
  if(button == "*")
  {
    button = "**";
  }
  var calc = Sys.Process("CalcPlus").Window("SciCalc");
  calc.Window("Button", button).Click();
}
```

3. The `checkCalcResult` function compares the expected result with that which is factually obtained:

```
function checkCalcResult(result)
{
  var calc = Sys.Process("CalcPlus").Window("SciCalc");
  var edit = calc.Window("Edit");
  aqObject.CompareProperty(edit.wText, cmpEqual, result);
}
```

4. By launching the `testExpectedValues` function, you will see that each expression among those assigned in the cells of the `Expression` columns has been calculated, and the result will be compared with the expected one (from the `Result` column).

### How it works...

First of all, we click on the **C** button to clear the results.

Further, the `testExpectedValues` function calculates one by one from each of the lines of the `Expression`, after which we obtain each symbol in the loop with the help of the `substr` method. This symbol is passed as a parameter to the `clickCalcButton` function.

The `clickCalcButton` function simply simulates clicks on the button with the passed heading. Exceptions arise only with the multiplication button: to click on the button with an asterisk, it is necessary to duplicate the same in the heading.

Then, with the help of the `checkCalcResult` function, we compare the obtained result with the expected ones.

Operations of clicks on a button and results verification are made into separate functions for better readability of the code and to avert code duplication.

## There's more...

Since in the Excel file the expected values have been inputted as numbers, the `agObject.CompareProperty` method transforms the text from the Calculator into an integer value. If the expected values had been placed into the Excel files as strings, we would have gotten an error for each of the cases in view, as the real value of the text field of the Calculator contains a point and a space at the end.

## Changing CSV delimiter and other parameters

By default in the CSV-files a comma is used as a separator for the fields.

In some instances however, it is more convenient to use another separator (for example, a semicolon, if the data often contains text with commas).

In this recipe we will consider changing the separators in the given CSV files.

## Getting ready

First of all, let's create a file and a script, which will be working with customizations by default:

1. Create a file `C:\data.csv` with the following content:

```
id,First,Last
1,John,Doe
2,Jane,Smith
```

2. Now, let's create a script, which will be reading the data from the given file:

```
function testCSVDelimiter()
{
    var data = DDT.CSVDriver("c:\\data.csv");
    while(!data.EOF())
    {
        Log.Message(data.Value("First"), data.Value("Last"));
        data.Next();
    }
    DDT.CloseDriver(data.Name);
}
```

## How to do it...

In order to have TestComplete read out the data that are separated by semicolon, it is necessary to complete the following actions:

1. Change the file `C:\data.csv` by changing commas to semicolons:

```
id;First;Last
1;John;Doe
2;Jane;Smith
```

2. Create the `C:\schema.ini` file with the following content:

```
[data.csv]
Format=Delimited(;
```

3. Launch the `testCSVDelimiter` function. In the result, the function will correctly read all the data, pre-separated by semicolon instead of the comma.

## How it works...

The file `schema.ini` is an ordinary INI file, which contains various parameters for CSV files in the same folder, where it is to be found. As a name of the section, the name of the file is used, and then their values are signified also. For example, in our case, we have denoted that data is to be separated by semicolon (the parameter `Format=Delimited(;`)).

This file can contain settings for several files that are to be found in the same catalogue.



Note that the name of the INI file should be exactly `schema.ini`.

## There's more...

Apart from the data separators, the `schema.ini` file can contain many more customizations (whether the file contains headings of the columns, the text encoding, data types for each of the columns, and so on). Reading up in greater detail about these settings is possible via the following link: <http://msdn.microsoft.com/en-us/library/ms709353.aspx>.

## Driving data without using loops

As in many examples (both in this book, and in the TestComplete documentation) to iterate through the data with the help of the DDT method, the `while` loop is usually applied.

This is not the only method to handle the DDT tables, and in this recipe we will consider another method, namely that of `DDTDriver.DriveMethod`.

### Getting ready

Create the file `c:\drivemethod.xls` with the following content:

	A	B	C	D
1	First	Last	Age	Position
2	John	Doe	36	Manager
3	Norma	Noe	30	Developer
4	Jack	Smith	32	Developer
5	Carla	Coe	25	Tester

### How to do it...

To go through all the records in the file without the loop, it is necessary to:

1. First create a function that will read all of the data from the currently handled column of the table:

```
function printPersonLastName()  
{  
    Log.Message(DDT.CurrentDriver.Value("Last"));  
}
```

2. Create the function, which will call the earlier written function: `printPersonLastName` with the help of the `DriveMethod` method:

```
function testDriveMethod()  
{  
    var data = DDT.ExcelDriver("C:\\drivemethod.xls", "Sheet1");  
    data.DriveMethod("Unit1.printPersonLastName");  
    DDT.CloseDriver(data.Name);  
}
```

3. If the function `testDriveMethod` is now be launched, the log will contain all of the names from the `First` column of the file `c:\drivemethod.xls`.

### How it works...

The `DriveMethod` method iterates through all the records in the current driver, each time applying the method, which was passed as a parameter to the `DriveMethod` method. The name of the launched method is passed wholly, because it has to contain the name of the module (in our case, the `Unit1`) and the name of the function itself (`printPersonLastName`). The type of the parameter is `string`.

To access the current parameters (properties and methods) of the DDT driver from within the evoked function, we use the `CurrentDriver` property.

Thus, with the help of the `DriveMethod` method, we have significantly simplified the code of the scripts, ridding ourselves of the loops and the need to non-forgetfully call the `Next` method at the end of the loop.

## Accessing Excel spreadsheets without having MS Office installed

If you try accessing the `DDT.ExcelDriver` method from your scripts without having MS Office installed, you will get the error **Provider cannot be found. It may not be properly installed.**

In this recipe we will learn how to avoid this problem without installing MS Office.

### How to do it...

In order to have access to Excel files via DDT perform the following steps:

1. Visit the following URL <http://www.microsoft.com/en-us/download/details.aspx?id=23734>.
2. Download and install **Data Connectivity Components** on your computer.
3. Now you can access DDT data from scripts.

### How it works...

Data Connectivity Components is a part of MS Office system, but it can be also installed separately to have access to `xlsx` files (files created in MS Excel 2007 or higher).

If you also need to edit Excel files on this computer, you can use the LibreOffice suite (<http://www.libreoffice.org/>) which is free.

## Auto-detecting Excel driver

The `DDT.ExcelDriver` method allows auto-detecting the Excel driver with the help of **Open Database Connectivity (ODBC)** or the Microsoft **Access Database Engine (ACE)** driver depending on the Excel file format after saving.

To this end, the `UseACEDriver` parameter is to be used. If you would not like to have these parameters signified explicitly each time, it is possible to write a wrapper function to do this automatically.

### How to do it...

To automatically define the type of the driver, we need to perform the following steps:

1. First we define the required `DDTExcel` function:

```
function DDTExcel(fileName, sheetName)
{
    var useACE = aqFileSystem.GetFileExtension(fileName) == ".xlsx";
    return DDT.ExcelDriver(fileName, sheetName, useACE);
}
```

2. Now we can dismiss the necessity to think about the type of the file as it will be defined automatically. The following two examples will work similarly:

```
var data = DDTExcel("C:\\data.xls", "Sheet1");
var data = DDTExcel("C:\\data.xlsx", "Sheet1");
```

### How it works...

The `useACE` parameter of the `ExcelDriver` method specifies which exact driver (ODBC or ACE) should be used to access file data.

To automatically define the type of the driver, we have used the simplest method of file-extension analysis. For the files with `xls` extension, the ODBC driver is used, while the driver of the type of ACE is utilized for the `xlsx` files.

### There's more...

It is possible to use the ACE driver both for the files of the `xls` type, and for the `xlsx` files as well. The following two examples will handle the task of file recognition equally effectively:

```
var data = DDT.ExcelDriver("C:\\data.xls", "Sheet1", true);
var data = DDT.ExcelDriver("C:\\data.xlsx", "Sheet1", true);
```

# 10

## Testing Web Applications

In this chapter we will cover the following recipes:

- ▶ Choosing Web Tree Model
- ▶ Using updates for the latest browser versions
- ▶ Performing cross-browser testing
- ▶ Verifying if a text exists on a page
- ▶ Waiting for an element to appear on a page
- ▶ Saving screenshots of an entire page
- ▶ Running scripts on a page

### Introduction

Nowadays, a lot of applications being developed are web based and most tools for automated testing have special abilities to test these types of applications. Such applications may be very complicated and include complex element structure and behavioral logic.

TestComplete supports testing web applications in all popular web browsers: Internet Explorer, Mozilla Firefox, Google Chrome, Apple Safari, and Opera. It also allows performing with web-based application all the things we can do with desktop software: search for elements, wait for elements, get pages' screenshots, and so on.

In this chapter we will discuss some of the frequently arisen topics related to testing web applications in TestComplete.



## Choosing Web Tree Model

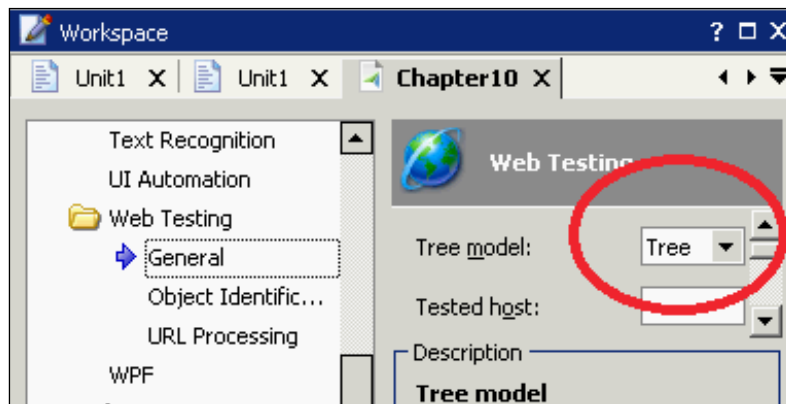
The structure of web-elements representation in **Object Browser** depends on the project setting **Web Tree Model**. This setting affects the way that web elements are displayed in **Object Browser** and accessed from scripts.

In this recipe we will consider the available web application models.

### How to do it...

To change the Web Tree Model settings, it is necessary to perform the following actions:

1. Right-click on the name of the project and navigate to the **Edit | Properties** menu item.
2. Open the following group of settings: **Open Application | Web Testing | General**.
3. In the **Tree model** drop-down list, select one of the elements (**Tree**, **DOM**, **Tag**, or **Hybrid**).

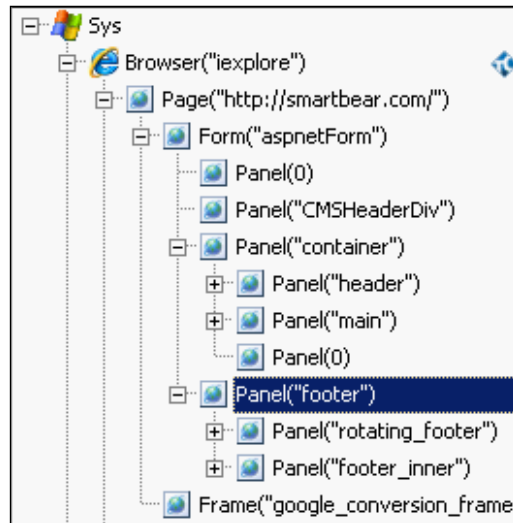


4. Save the changes by pressing **Ctrl + S**.
5. The controls elements in the browser will be shown differently, depending on the selected model. Accessing those from the scripts will be different as well.

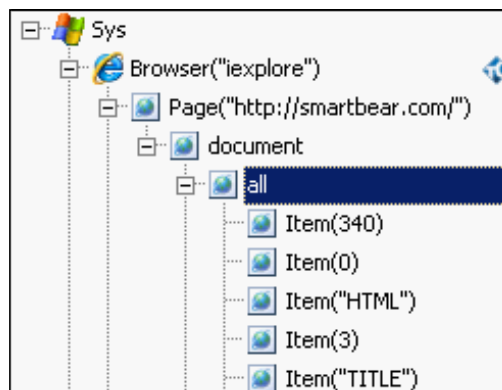
## How it works...

TestComplete supports the following models for web applications:

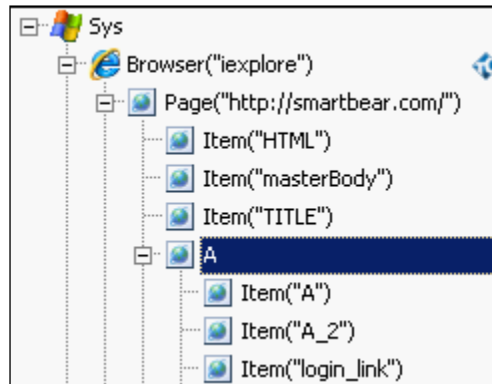
- ▶ **Tree:** This is the recommended model (in TestComplete 9 all the other models are considered as obsolete and necessary only for backward compatibility with the previous versions). When the `Tree` is being used, the hierarchy of control elements in **Object Browser** corresponds with that of the application. The `Tree` is the only model with cross-browser testing support.



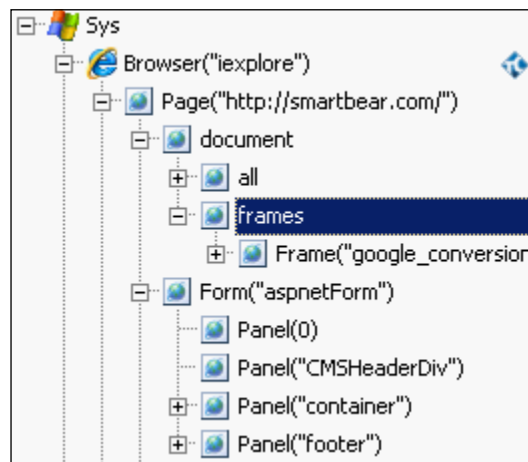
- ▶ **DOM:** In this model all the webpage elements are available as siblings of the `document.all` element. If the page contains frames, they can be accessed via the `document.frames` object, similarly to the way they are represented in the DOM model of the webpages.



- ▶ **Tag:** When using this model, the control elements are grouped by tags in the elements with the corresponding names. The elements that are found to be singular (for example, HTML, BODY, and so on) do not have sibling elements; rather they directly correspond to the eponymous page elements.



- ▶ **Hybrid:** This is a combination of models `Tree` and `DOM` that is meant for creation of the new scripts with the use of the recommended `Tree` model, simultaneously lending support for the older scripts which utilized the `DOM` model:



If you are not content with the recommended `Tree` model, you can try using other models; however, the cross-browser `TestComplete` testing possibilities will not be available.

## There's more...

If you have some older scripts, written with the help of the `Tag` model, and you're willing to write some new ones using the recommended `Tree` model, you could dynamically signify which model to use for each of the tests. An instance of dynamically changing the objects model from script is shown in the following code snippet:

```
Options.Web.TreeModel = "Tag"
Options.Web.TreeModel = "Tree"
```

## See also

- ▶ You can read more about cross-browser testing in the *Performing Cross-browser testing* recipe in this chapter

## Using updates for the latest browser versions

Every TestComplete release supports a certain browser version (the latest version at the release date). In this recipe, we will consider how to download and install support for newer browser versions.

## How to do it...

In order to add support for a new browser version we need to perform the following steps:

1. Navigate to the following URL: <http://support.smartbear.com/downloads/testcomplete/>.
2. Click on the link with corresponding browser name (for example, **Firefox Patches**).
3. Download the archive which corresponds to your TestComplete version and necessary browser version (for instance, Firefox 24, TestComplete 9.31).
4. Unpack the archive content and open the file `Installation_Notes.htm`.
5. Follow the instructions from the file.

## How it works...

Since it is impossible to guarantee support for the future browser versions, SmartBear publishes patches once a new version of browser is released. Creating such a patch may take some time, therefore it is recommended to disable automatic updates in your browser, otherwise your regular script runs may be affected while waiting for the patch release.

It is also recommended that you create reserve copies of the files which you replace when applying a patch.

## Performing cross-browser testing

TestComplete provides a possibility to perform Web-applications testing in all of the popular browsers (Internet Explorer, Mozilla Firefox, Google Chrome, Apple Safari, and Opera), easily switching in-between.

In this recipe we will consider launching one script in several browsers.

### Getting ready

With the help of recording means let's make a record of any test in any of the available browser at hand. In our example, Internet Explorer is used to carry out the following actions:

1. In TestComplete go to **Test | Record | Record Script** menu item.
2. Open the browser.
3. Navigate to the following page: <http://smartbear.com/>.
4. Click on the **Free Trial** link.
5. In the result, the following script will be recorded:

```
function Test1()
{
    var form;
    Browsers.Item(btIEExplorer).Run("http://smartbear.com/");
    browser = Aliases.browser;
    page = browser.pageAutomationTestingWebMonitori;
    page.formAspNetform.linkLookingForAFreeTrial.Click();
    browser.pageSoftwareTestingWebMonitoring.Wait();
}
```

### How to do it...

To launch the recorded script in several browsers, go about the following actions:

1. Create a new test with the following code:

```
function Test2()
{
    var browsers = ["iexplore", "firefox"];
    for(var i = 0; i < browsers.length; i++)
    {
        //TODO: place code here
    }
}
```

2. Remove the `//TODO` comment and place the contents of the earlier written `Test1` function instead.
3. Replace the line `Browsers.Item(btIExplorer)` with the following: `Browsers.Item(browsers[i])`. In the result, we will have the next function ready:

```
function Test2()
{
    var browsers = ["iexplore", "firefox"];
    for(var i = 0; i < browsers.length; i++)
    {
        var form;
        Browsers.Item(browsers[i]).Run("http://smartbear.com/");
        browser = Aliases.browser;
        page = browser.pageAutomationTestingWebMonitori;
        page.formAspnetform.linkLookingForAFreeTrial.Click();
        browser.pageSoftwareTestingWebMonitoring.Wait();
    }
}
```

4. By launching the `Test2` function, we will see that the same actions were initially executed in the Internet Explorer browser, and then in Mozilla Firefox.

### How it works...

The object `Browsers` provides access to any installed browser via the `Item` property. This property takes the `Index` parameter, with the help of which we are able to assign the particular browser we would like to work in. The `Index` parameter can be any number (of a named constant value, just like the `btIExplorer` in our case), or the name of a process.

Since working with control elements in all the browsers is executed in a similar fashion, we can record a script in one browser to execute in another or in several others (in our example, we have used the `for` loop).

In a real project, it is more convenient to create tests that accept a parameter (the name of browser process), and then launch the tests by passing the name of the browser as a parameter. The resulting function will assume the following shape and form:

```
function Test3(browserName)
{
    var form;
    Browsers.Item(browserName).Run("http://smartbear.com/");
}
```

The launch of this from the test items will appear as follows:

Name	Test	Parameters
<input checked="" type="checkbox"/> TestIE	Script\Unit1 - Test3	browserName (Value = iexplore)
<input checked="" type="checkbox"/> TestFF	Script\Unit1 - Test3	browserName (Value = firefox)

## See also

- ▶ Although TestComplete does provide wide possibilities for cross-browser testing in some cases, however, browsers may behave differently, thus requiring additional settings and code. If you create a lot of cross-browser tests, make sure you read up on the following article: *Handling Browser Differences*, located at the following address: <http://support.smartbear.com/viewarticle/27716/>.

## Verifying if a text exists on a page

Sometimes, to check if the targeted webpage has been completely opened (instead of receiving an instance of the 404 page not found error), it is enough to see if the page contains some specific text (usually, the text is unique for the given page).

In this recipe we will consider an example of such verification.

## Getting ready

Launch the Internet Explorer and open the following site: <http://smartbear.com/>.

On the main page, locate any text, which will be unique namely for the given page (in our example, we will be using the **Tools for your needs** text).

## How to do it...

To check if the targeted text is available on a page, it is necessary to perform the following actions:

1. Write the `isTextPresent` function that executes the verification:

```
function isTextPresent(element, text)
{
    return aqString.Find(element.contentText, text) > -1;
}
```

2. Let's now write a simple script, using the `isTextPresent` function (replace the highlighted text with the one you have located in the beginning of this recipe):

```
function testVerifyText()  
{  
    var br = Sys.Browser("iexplore");  
    var page = br.Page("http://smartbear.com/");  
    var form = page.Form("aspnetForm");  
    Log.Message(isTextPresent(form, "Tools for your needs"));  
}
```

3. If you launch the `testVerifyText` function, the result in the log will be **True**.

### How it works...

The `contentText` property contains the whole text of the element, including related child elements. This property is analogous to those of `innerText` and `textContent`; however, unlike those, it works equally in all the browsers.

In our example, we have used the `contentText` property for the whole of the page (that is, for the element that corresponds to the page), which means we have obtained all the available text on-site.

Similarly, we can obtain all the text of any other web-element (that is, `Panel` or `TextNode`).

## Waiting for an element to appear on a page

In the majority of cases, to determine the completion of the page download, we can use the `wait` method. However, in some cases, download completion is determined by appearance of a specific element (occurring after the page download is flagged as completed).

In this recipe we will consider some methods to bind handler-functions to an event, signaling appearance of a certain webpage element. As an example, we would make use of the **Login** link on SmartBear main page.

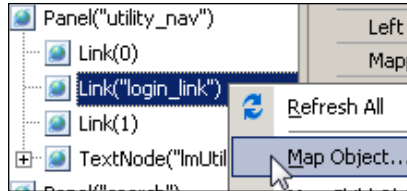
### Getting ready

Before we turn to waiting for the **Login** link to appear straight off the bat, we can make some simple preparations to simplify the task at hand:

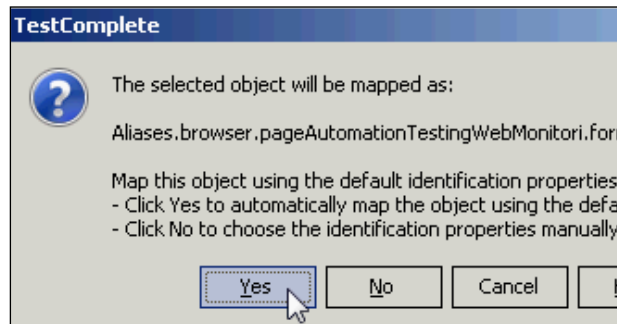
1. Open the main page of SmartBear company (<http://smartbear.com/>) in your browser (in our examples, we are using Internet Explorer).



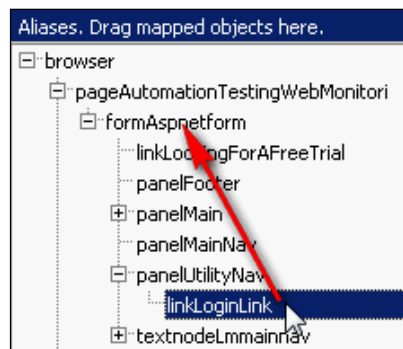
- In TestComplete, open **Object Browser** and locate an element therein, which would correspond to the **Login** link (to this end, you could make use of the **Object Spy** tool).
- Right-click on this element and opt for the **Map Object** menu item.



- In the opened window, click on **Yes** to automatically add the element with the given properties by default.



- Open the element of the **NameMapping** project and locate the created **linkLoginLink** element therein.
- With the mouse, drag-and-drop the targeted element onto the element of **formAspnetform**. Now, the link has become a sibling element of the page.



- We have additionally renamed the control elements by giving them succinct names for brevity and convenience's sake (pageMain, formMain, and so on).

## How to do it...

To bind to the `linkLogin` element, we will use the `waitAliasChild` method:

1. Let's write the following code, which will be waiting for the link to appear:

```
var br = Aliases.browser;
br.ToUrl("http://smartbear.com");
var page = br.pageMain;
var form = page.formMain;
var login = form.WaitAliasChild("linkLogin", 5000);
```

2. Now we will additionally check if the element is available on the page:

```
if(!login.Exists)
{
    Log.Error("Login link didn't appear on the page");
}
```

3. If you were to evoke this function again, it would transfer onto the `SmartBear.com` site and will wait for the **Login** link to appear.

## How it works...

The `waitAliasChild` method allows waiting for the appearance of a sibling element within the preset timeout, assigned by the second parameter of the method (in our case, we are dealing with 5000 milliseconds or 5 seconds).

This method returns the object, whose `Exists` method contains the information, whether the element has made it to the webpage or not. In case the element fails to show up, the `Exists` property will contain the `False` value, and we will see the error message in the log.

Such a method of handling web-pages is usually used at working with applications, where AJAX is employed, since some of the control elements may have extended latency, compared to that of the main page.

## There's more...

Along with the `waitAliasChild` method, there also exists a similar method of `waitNamedChild`, allowing one to wait and observe the element, passing its name as a parameter from **NameMapping** instead of the **Aliases**.

## See also

- ▶ Working with the `wait` methods is dealt with in greater detail in the *Waiting for an object to appear* recipe from *Chapter 5, Accessing Windows, Controls, and Properties*

## Saving screenshots of an entire page

Some of the webpages exceed the height of the browser and require use of scrollbars to view the contents in the bottom of the page.

This influences creation of screenshots of the pages: concealed contents are not visible at the point of making a screenshot via the `Picture` method.

In this recipe we will learn how to create screenshots regardless of the dimensions of the window in the browser.

### How to do it...

To create the screenshot of the whole of the webpage:

1. Let's write a script to open the main page of the site of the SmartBear company:

```
var br = Aliases.browser;
br.ToUrl("http://smartbear.com");
var page = br.pageMain;
```
2. With the help of the `PagePicture` method we will save the screenshot of the page to the log:

```
Log.Picture(page.PagePicture());
```
3. If you launch the example, you will see the whole of the page saved to the log, including the section whose window-overflow parts were hidden.

### How it works...

The method of `PagePicture` automatically scrolls the page making screenshots of its parts, and then automatically splices them to make a screenshot of the whole of the page.

Please note that, unlike the `Picture` method, the `PagePicture` method creates the screenshot of the page alone, and not that of the whole screen. This means: if there's a window on screen at the moment the screenshot is made (for example, an error alert that impacts work of the tested application), the screenshot will not show the window.

### See also

- ▶ More on screenshots creation could be learned from the *Posting screenshots to the log* recipe [Chapter 6, Logging](#)

## Running scripts on a page

Sometimes, when testing web applications, there arises a need to launch some script on the page.

The easiest way to do so is launching the script directly from the address bar of the browser.

### How to do it...

To launch the script in the browser, it is necessary to:

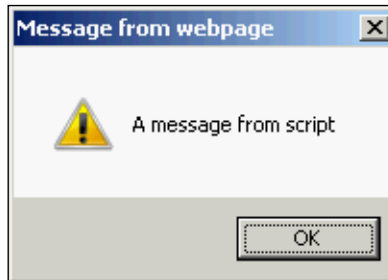
1. First of all, write the script which transfers current URL location to a specific address:

```
var br = Aliases.browser;  
br.ToUrl("http://smartbear.com");  
var page = br.pageMain;
```

2. Now we will launch the script that will execute JavaScript code on the page:

```
page.NavigateTo("javascript: alert('A message from script');");
```

3. In the result, the browser window will have the following message appear:



### How it works...

To execute the JavaScript code in the browser window, it is sufficient to input the code into the address bar of the browser as a string, placing the keyword `javascript:` before the pasted code.

The method `NavigateTo` lets us open the page specified by the `URL` parameter. Unlike the `ToUrl` method, method `NavigateTo` doesn't return any value after the page is loaded.

In our case, the method of `NavigateTo` is used to execute the JavaScript code on the page.

## See also

- ▶ Some browsers do not support the described method of launching scripts. If you have come up against such a problem, you could implement the code of the script to the opened page and then execute the same. This method is described in the article *Embedding Test Scripts Into Web Pages* on the SmartBear website. (<http://support.smartbear.com/articles/testcomplete/embedding-scripts-into-web-pages>).

# 11

## Distributed Testing

In this chapter we will cover the following recipes:

- ▶ Setting up Network Suite and understanding distributed testing
- ▶ Copying Project Suite to a Slave workstation
- ▶ Using a Master workstation to run tests
- ▶ Using different configuration files for each workstation
- ▶ Sharing data between workstations
- ▶ Synchronizing test runs on several workstations

### Introduction

Sometimes, acceptance testing is performed on several workstations. There may be different reasons for this: time required for running all tests is too much to run whole Project Suite on one computer or we might need to run tests on computers with different operating systems, and so on.

In any case, it would be nice to be able to run and control all these tests from a single computer, and then to be able to collect and analyze test results in one place.

TestComplete provides us with a feature called Network Suite which allows running tests on several workstations simultaneously and yet controls the execution flow from a single place.

In this chapter, we will discuss some frequently asked questions about Network Suite usage.

## Setting up Network Suite and understanding distributed testing

Distributed testing is applied against impossibilities to launch and to run tests on the same computer. For example, if the total working time of the scripts is up sky-high or it is necessary to test simultaneously via different configurations, the distributed testing is necessary.

In this recipe, we will consider an example of making the necessary customizations for the distributed testing.

### How to do it...

To set the stage for distributed testing, it will take two computers: the local one (hereafter, the **Master** computer) and the remote one (later referred to as **Slave**):

1. Install TestComplete or TestExecute on the Slave computer. The versions of the applications on the Master and Slave computers should be the same.
2. Add the **Network Suite** element to the main project of the current Project Suite (right-click on the name of the project and select **Add | New Item**). The main project is highlighted in bold font in the **Project Explorer** tab.
3. Add the new **Host** computer to the **NetworkSuite** element (expand the **NetworkSuite** element, right-click on the **Hosts** element, and select **Add | New Item...**). Now, click on **OK**.
4. Double-click on the added host **Host1** and input the parameters for the Slave computer in the right-hand side section of the TestComplete window:
  - **Address:** Enter the name of the computer or its IP address
  - **Login Mode:** Choose the **[Manual]** option from dropdown
  - **Domain, User name, and Password:** Enter account parameters for the Slave computer
  - **Base Path:** Enter the path on the Slave computer to clone the current project to

Name	Address	Login mode	Domain	User name	Password	Base path
Host1	TOSHIBA ...	[Manual]		user	*****	C:\Tests

5. Right-click on the edited host and select the **Copy Project to Slave** option. The Slave computer is now ready to work. Let's get the Master computer customized for the launch.
6. Right-click on the **Jobs** project element, select the **Add | New Item...** option, and, click on **OK**.

7. Now, right-click on the **Job1** created element, select the **Add | New Item...** option, and click on **OK**.
8. Double-click on the created **Task1** task and, in the right-hand side part of the TestComplete screen, input its parameters as follows:
  - ❑ **Name:** Enter the name of the task
  - ❑ **Host:** Enter the name of the earlier created host, `Host1`, in our example
  - ❑ **Remote application:** Enter the name of the application, installed on the Slave computer
  - ❑ **Test:** Enter the name of the test which is bring launched

Active	Name	Host	Remote application	Test
<input checked="" type="checkbox"/>	Task1	Host1 ...	[TestExecute]	Project1\Script\Unit1\Main ...

9. Now, it is possible to launch the selected test on the Slave computer. To this end, on the Master computer, right-click on the **Task1** element and select the **Run** option.
10. In the result, the Slave computer will run the test items of the project and the generated report in the MHT format will be opened; the whole of the process will be controlled and displayed on the Master computer.

## How it works...

In distributed testing, the two types of computers that participate are:

- ▶ **Master:** This is the main computer that will launch and control all of the launches
- ▶ **Slave:** These are the computers that have the tests performed on them

When cloning a local project onto the Slave computer, the project folder will get copied by default. If it's necessary to copy some other folder, you should signify the same in the **Source path** field of the host. If, by default, the **Source path** column is not visible, it can be added by right-clicking on the columns' names, having selected the **Field Chooser** option, and then dragging-and-dropping the name of the column onto the heading of the hosts' tables.





**Jobs** is a list of tasks that are executed simultaneously on several computers. Besides the parameters considered earlier, each job is comprised of some additional parameters which are given as follows:

- ▶ **Copy remote log:** This is a condition which specifies when the log from the Slave computer should be copied onto the Master computer (always, never, or in case of errors)
- ▶ **Action after run:** This is the action upon the end of the launch (closing the application, reboot, or shut the Slave computer down)
- ▶ **Use previous instance:** This is the action against attempts to launch the application (TestComplete or TestExecute) when it is already up-and-running (use the available one, re-launch the application, or notify of the error)

### There's more...

In some cases, you might need to perform additional actions to be able to use Network Suite. For example, configure Windows Firewall if it is enabled or make changes to an antivirus program. You can find the requirements for Distributed Testing using this link: <http://support.smartbear.com/viewarticle/27241/>.

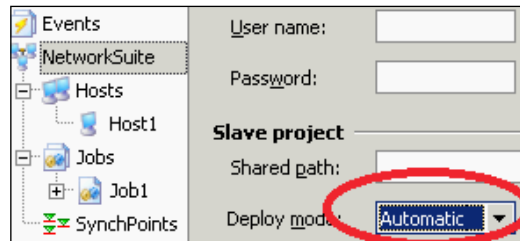
## Copying Project Suite to a Slave workstation

Copying a project into the Slave computer manually is not that practical (since you would have to keep doing so regularly without forgetting), this is why this process would be better to automate. In this recipe, we will consider how to do this.

### How to do it...

To automatically copy the project into the Slave computer, it is necessary to perform the following actions:

1. Double-click on the **NetworkSuite** projects element.
2. In the right-hand side part of the TestComplete window, in the **Deploy mode** drop-down list, select **Automatic**.



3. Save the changes (for example, pressing the **Ctrl + S** keys).

## How it works...

When the distributed testing starts, TestComplete automatically copies the projects from the Master computer into the Slave computer. The source folder and the destination folder for copying is to be signified in the **Source path** and **Base path** input fields of the host respectively. If nothing is signified in the **Source path** field, TestComplete will copy the current project into the destination folder.

## There's more...

Copying the project is also possible from the scripts, invoking the `CopyProjectToSlave` method for the corresponding host, for example:

```
var host = NetworkSuite.Hosts.ItemByName("Host1");
host.CopyProjectToSlave();
```

## Using a Master workstation to run tests

TestComplete does not allow adding a Master computer to the hosts listing (thus, using the same as a Slave computer); however, the possibility to launch and run tests on the Master computer is still there.

## How to do it...

To launch the tests simultaneously on the Slave and Master computers, it is necessary to go about the following steps:

1. Start a job on the Slave computer:

```
NetworkSuite.Jobs.ItemByName("Job1").Run(false);
```

2. Then launch the tests on the local computer:

```
test1();
```

```
test2();
```

```
testN();
```

3. Finally, wait for the tests to complete on all the Slave computers:

```
NetworkSuite.WaitForState(ns_Idle);
```

## How it works...

The only parameter of the `Run` method is the `WaitForCompletion` parameter that is preset to the value of `False`, which allows launching the tests on the Slave computers and carries on with the script's execution.

Further on, we launch any tests locally, after which using the `NetworkSuite.WaitForState` method we wait for the script's execution to come through on all the Slave computers. Without this method's invocation, execution of the tests on the Slave computers will terminate, if the execution on the Master computer completes beforehand.

## Using different configuration files for each workstation

When testing on several different computers, there sometimes arises a necessity to use various testing data or parameters. It may depend on the version of the operational system, its bit-rate, or it can even be different for every computer.

The simplest method to use different testing data in such cases is having data sets for each particular case—each time reading the data from the appropriate file.

## Getting ready

Let's suppose, we have a file `settings.ini`, that contains all the preset parameters for the tests, to be declared in the code as follows:

```
var file = "settings.ini";  
// code for settings read
```

At the point of launching tests on various operational systems, it turned out that each OS requires different settings.

## How to do it...

To read supply settings from different files depending on the OS version, it is necessary to perform the following steps:

1. Make several copies of the `settings.ini` file with the following names: `winXP.ini`, `winVista.ini`, `win7.ini`, and `win8.ini`.
2. Write parameters in each file that are necessary for the eponymous OS.
3. Change the code for the variable declaration in the following manner:

```
var file = Sys.OSInfo.Name + ".ini";
```

- Now, the settings will be read from different files, depending on the Windows version used to launch and run the scripts.

### How it works...

The `OSInfo.Name` parameter returns a short name of the current operational system. The unbridged name of the OS is possible to obtain from the `OSInfo.FullName` property.

Similarly, it is possible to organize reading the data for other cases of the kind. For example, if the data differs for every computer, the file names can be given by computer name, while variables could be created via the `Sys.HostName` property.

In case of the **Data-driven Testing (DDT)** approach, it is possible to store the data for different operational systems on different pages of the document, each corresponding to the proper operating system.

## Sharing data between workstations

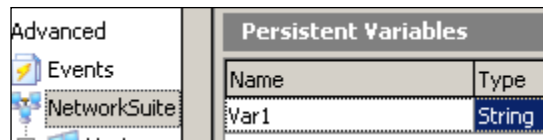
To exchange the data between the Slave computers, the Network Suite variables are designed to make this available at any given moment on any Slave and the Master computer.

In this recipe, we will deal with an example of data exchange with the help of Network variables.

### Getting ready

First of all, we need to create the Network variable. To this end:

- Right-click on the **NetworkSuite** project element and select the **Edit | Variables** option.
- Right-click on the **Persistent Variables** area and select the **New Item** option. In the result, a new variable `Var1` of the `String` type will appear on the list.



## How to do it...

To change the value of the `Network` variable, it is necessary to accomplish the following steps:

1. First of all, enter the code critical section:

```
NetworkSuite.EnterCriticalSection("Change Variable");
```

2. Change the value of the variable:

```
NetworkSuite.Variables.Var1 = "NEW VALUE";
```

3. And finally, exit the critical code section:

```
NetworkSuite.LeaveCriticalSection("Change Variable");
```

4. Now, the `NetworkSuite.Variables.Var1` variable contains a new value, which is accessible to any computer for the distributed testing.

5. It is possible to write a function that will do so, automatically:

```
function changeNetworkVar(varName, value)
{
    NetworkSuite.EnterCriticalSection(varName);
    NetworkSuite.Variables.VariableByName(varName) = value;
    NetworkSuite.LeaveCriticalSection(varName);
}
```

6. Now, changing the variable will look as follows:

```
changeNetworkVar("Var1", "NEW VALUE");
```

## How it works...

The variables of the `NetworkSuite` level are accessible for the scripts on all the computers throughout the distributed testing.

Usage of the code critical sections allows avoiding conflicts while simultaneously changing one and the same variable from different locations, all at once. As soon as one of the projects starts the critical code section, other projects will not be able to get on with the same critical code section under the same name, and would be waiting for its closure. This is why, to change one variable, it is always necessary to use the same names of the code critical sections.

When reading the values of the variables, the earlier mentioned conflicts will not arise, and this is why, for reading the values, we don't have to use code critical sections.

Usage of the code critical sections for changing variables is not mandatory; however, it is strongly recommended to preclude the synchronization errors. Locating the reasons of such errors is quite a task.

## There's more...

Another useful possibility to work with the `Network` variable is waiting for the variable to assume the assigned value. To this end, the `WaitForNetVarChange` method is used:

```
NetworkSuite.WaitForNetVarChange("Var1", "NEW VALUE", 10000);
```

This example shows the variable `Var1` is set to wait for maximum 10 seconds to assume the `NEW VALUE` value. If the last parameter is set equal to zero, the waiting would be endless.

This method would help to synchronize tests execution on different Slave computers.

## See also

- ▶ The variables of the `NetworkSuite` level are analogous to the Project variables, which are dealt with in the *Using global variables* recipe in *Chapter 3, Scripting*.

## Synchronizing test runs on several workstations

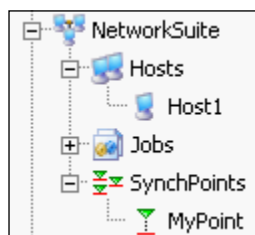
Sometimes, there arises a need to synchronize tests launches from various projects to have specific actions executed at the same time. Usually, it is necessary when we perform concurrent testing or testing workflows which require simultaneous actions of multiple users.

To this end, synch points are applied, which will be thoroughly dealt with in this recipe.

## How to do it...

To synchronize tests on several Slave computers, we need to perform the following steps:

1. Expand the **NetworkSuite** element, right-click on the **SynchPoints** element, and opt for the **Add | New Item...** option.
2. In the opened **Create Project Item** window, enter the name of the synch point (for example, `MyPoint`), and click on **OK**.



3. Repeat the steps for all the projects that needs to be synchronized.
4. Now, add the following code in the code earmarked for synching:

```
NetworkSuite.Synchronize("MyPoint");
```

### How it works...

At the point of distributed tests launch, when reaching the synching points, the tests will pause; continuation of the tests will take place only after all the tasks have reached the synch point.

In the case that a project contains a synch point, and at least one of the projects fails to evoke the `NetworkSuite.Synchronize` method, this would lead to script hang-ups, as the scripts would be waiting for all the projects to reach the synch point in view. In this case we will have to stop test execution manually.

This is why it is recommended that we attentively follow through the `NetworkSuite.Synchronize` invocation, if the project contains synch points.

### See also

- ▶ Another method for synchronization with the help of the variables is described in the *Sharing data between workstations* recipe.
- ▶ All the synchronization variants are given an in-depth description in the *Synchronizing Distributed Tests* article, located on the SmartBear website: <http://support.smartbear.com/articles/testcomplete/synchronizing-distributed-tests>.

# 12

## Events Handling

In this chapter we will cover the following recipes:

- ▶ Creating event handlers
- ▶ Disabling the postage of certain error messages
- ▶ Clicking on disabled controls without an error message
- ▶ Handling unexpected windows that affect TestComplete
- ▶ Handling unexpected windows that don't affect TestComplete
- ▶ Saving the log to a disk after each test
- ▶ Sending a notification e-mail on timeouts
- ▶ Creating preconditions and postconditions for tests

### Introduction

Every development environment allows handling events, that is, creating custom actions for different situations an application may face during its work.

TestComplete is also a development environment with its own events, and it also allows us to create handlers for different events: Log events, Network Suite events, general events of test engine, and so on.

In this chapter, we will consider the most frequently used events handlers one may need to know when performing test automation with TestComplete. We will also consider some tasks which can be best solved with the help of event's handlers.



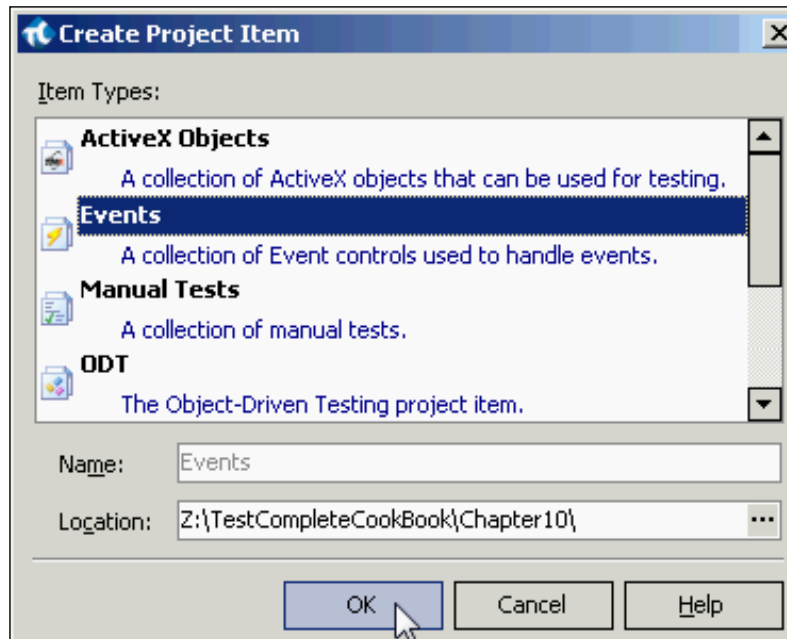
## Creating event handlers

In order to operate with events, we need to learn how to create event's handlers in TestComplete. This recipe will guide you on creating an event handler for the `OnLogError` event (this event fires every time error is posted to the TestComplete log).

### Getting ready

Before creating the event handler, we need to add the **Events** project item if it is not added to project yet:

1. Right-click on the project name in the **Project Explorer** tab and navigate to **Add | New Item**.
2. In the opened **Create Project Item** window, select the **Events** item and click on **OK**:

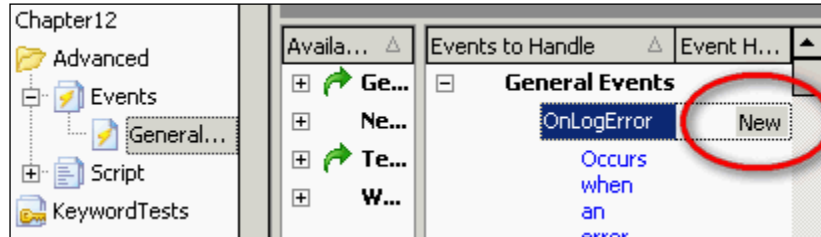


### How to do it...

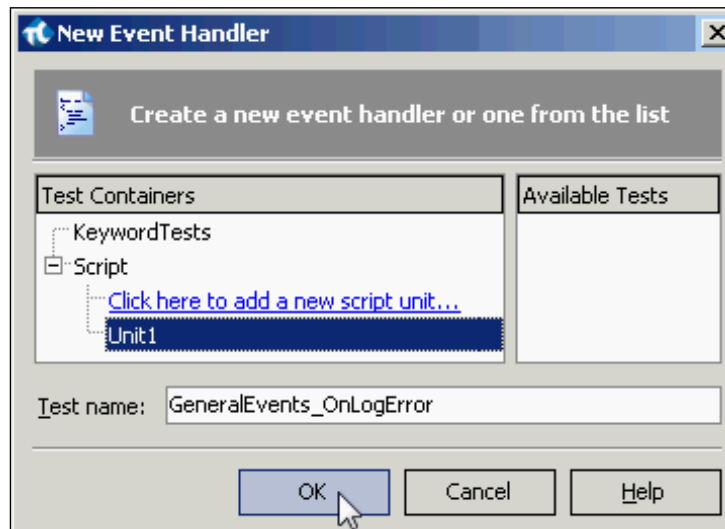
To create the event handler for the `OnLogError` event we need to perform the following steps:

1. Expand the **Events** node in the project and double-click on the **General Events** item.
2. In the **Events to Handle** list, expand the **General Events** node and select the `OnLogError` item.

- Click on the **New** button which is located inside the OnLogError element:



- In the **New Event Handler** window, select the script unit where you want to create the handler and click on the **OK** button:



As a result, a new function `GeneralEvents_OnLogError` will be created in the selected unit. This function will fire every time some is posted to the log and all actions from it will be executed.

### How it works...

It is not enough to just create a new function manually with corresponding name (for example, `GeneralEvents_OnLogError`) to create an event handler. We have to perform all the previous steps described to accomplish the task.

When selecting a script unit for storing event handlers, we can create a new one by clicking on the **Click here to add a new script unit...** node inside the **Script** node. It is recommended to store all events in one separate unit for easier maintenance.

We can also use an existing function as an event handler by selecting its name in the **Available Tests** list in the **New Event Handler** window.



Please note that event's handlers should have exactly the same parameters as they are expected by TestComplete, that's why it is better to create event's handlers by TestComplete means.

### There's more...

To remove an existing event handler (not the function itself), right-click on its name in the **Events to Handle** list and select menu item **Remove Event Handler**.

If you want to completely remove the function, simply remove it from the unit in the editor.

## Disabling the postage of certain error messages

Any message in the log can be captured and processed. In this recipe we will consider the following example. When interacting with the `Infragistics` controls elements in the TestComplete log, there sometimes arises a message **Improper command**. This message in no way influences the script's performance and only causes an issue when reviewing log entries; this is why we will simply ignore it.

This is a true-to-life example, which is successfully applied in a commercial project.



To try this example, you can simply create a function with several calls of the `Log.Error` method. At least one of these calls should contain **Improper command** text as an error message.

### How to do it...

To avoid logging of the **Improper command** message, it is necessary to perform the following actions:

1. Create an `OnLogError` event handler.
2. Interject the following code into the event handler:

```
if(LogParams.MessageText == "Improper command")
{
    Log.Message("'Improper command' error is ignored");
    LogParams.Locked = true;
}
```

Now, all the errors with the `Improper command` text will be ignored.

## How it works...

The `LogParams` object contains all the information on the outputted message to the log.

The `MessageText` parameter contains the text of the message, and the `Locked` parameter, preset to the `true` value is blocking the entry of the message to the log. This block only affects the current event, not the others.

Simple ignoring the errors is a not a good practice, this is why we additionally notify in an ordinary message what exactly is being blocked. If, in the future, due to this error, there emerges any technicalities with the control elements, we will be able to spot these issues in the log and thus will be in the know for the root-cause analysis.

Similarly, other events of the log are to be handled (`OnLogEvent`, `OnLogMessage`, and so on).



In the earlier versions of TestComplete (up to 7 inclusively), instead of the `MessageText` property, the `Str` property was put to use. Their inner workings are the same; however, the `Str` property (and the `StrEx` one, corresponding to that of `AdditionalText`) are now considered as deprecated and are not recommended for use in the new scripts.

## See also

- ▶ The process of creation of the event handler is thoroughly dealt with in the *Creating event handlers* recipe

## Clicking on disabled controls without an error message

If you need to trigger mouse-clicks on a disabled controls element, recording such a script would be a pushover; however, when executing the script, TestComplete will output the following error message to the log: **The window is disabled. The action cannot be executed.**

In this recipe, we will consider emulating a mouse-click on the disabled controls elements, following an example of button **A** in the calculator.

## Getting ready

Launch the Calculator Plus application in the Scientific mode (navigate to **View | Scientific**).

**How to do it...**

To click on the disabled controls element, it is necessary to perform the following steps:

1. First of all, let's write up some code to trigger a mouse-click on the **A** button (executing this code will lead to the previous discussed error in the log):

```
function testClickDisabledButton()
{
    var wCalc = Sys.Process("CalcPlus").Window("SciCalc");
    wCalc.Window("Button", "A").Click();
}
```

2. Now, let's write up a function that will trigger a mouse-click on a disabled controls element:

```
function clickDisabledObject(obj)
{
    var x = obj.ScreenLeft + obj.Width/2;
    var y = obj.ScreenTop + obj.Height/2;
    Sys.Desktop.MouseDown(VK_LBUTTON, x, y);
    Sys.Desktop.MouseUp(VK_LBUTTON, x, y);
}
```

3. Now, create an event handler for the OnLogError event.
4. Interpolate the created event handler into the following code:

```
if(aqString.Find(LogParams.MessageText, "The window is
disabled") > -1)
{
    Log.Message("The window is disabled");
    var objText = LogParams.AdditionalText.split("\n")[1];
    var obj = eval(objText);
    clickDisabledObject(obj);
    Log.Event("Disabled object was clicked", objText);
    LogParams.Locked = true;
}
```

5. If we launch the `testClickDisabledButton` function again, clicking on the **A** button will be reproduced, and the log would contain two messages posted by using the `Log.Event` and `Log.Message` methods. There would be no errors in the log.

## How it works...

Standard `TestComplete` means do not allow for a mouse-click on the disabled controls element: this is why, to resolve the task, we have made use of these methods: `Sys.Desktop.MouseDown` and `Sys.Desktop.MouseUp`. These two methods trigger the mouse-click and release the assigned mouse-button (in our case, the left one that is, **VK\_LBUTTON**), in the assigned screen coordinates ( $x$  and  $y$ ). To obtain the screen coordinates of the center of the object, we have used the following properties: `ScreenLeft`, `ScreenTop`, `Width`, and `Height`.

In the event handler itself, we are checking if the error message contains the following string: **The window is disabled**. If so, we will get an object targeted for the mouse-click, and pass it as a parameter to the `clickDisabledObject` function.

Finally, with the help of setting the property `LogParams.Locked = true`, the error output to the log is blocked.

The process of obtaining the object needs to be explained in more detail. Since we cannot get the object directly from any of the properties of the `LogParams` object, we have to retract it from the `AdditionalText` property. First, via the `split` method, we obtain the second string of the message (this string contains the complete name of the controls element), and then with the help of the `eval` function, the text is further transformed to the object (the `eval` function transforms the string into the executable JScript code).



Pay attention that all the executable actions are being logged (with the help of the `Log.Message` and `Log.Event` methods). This is a good style of writing similar event handlers as well as any other generic code, since further on, when analyzing the logs, this information might come in handy.

## See also

- ▶ The process of creation of an event handler is dealt with in greater detail in the *Creating event handlers* recipe

## Handling unexpected windows that affect TestComplete

An unexpected window is a window which shows up as scripts are being executed, affecting TestComplete interaction with tested application. In the project settings, it is possible to specify what should be done with such windows (whether they should be ignored, paused, or stopped; or a number of simple actions could occur); however, in some cases, it is convenient to handle some of the windows (known to us) independently.

In this recipe, we will consider an example of working with the **Print** window of the Notepad application. To simulate unexpected appearance of the window, we would be evoking it ourselves, pressing the *Ctrl + P* keys combination. Our script would input the text into Notepad and, upon the **Print** window appearing, enter a corresponding message to the log about the event; and then simply wait for the window to close out. Closing the window is also expected to be done manually.

### Getting ready

Launch the standard Windows Notepad (`C:\Windows\notepad.exe`) application.

### How to do it...

To create the **Print** window handler, it is necessary to perform the following steps:

1. First of all, let's create a script which will execute some actions (for example, type a text) in the Notepad:

```
function testUnexpectedWindowHandling1()
{
    var np = Sys.Process("NOTEPAD").Window("Notepad");
    np.Activate();
    for(var i = 1; i < 1000; i++)
    {
        np.Keys(i + "[Enter]");
    }
}
```

2. Create the `OnUnexpectedWindow` event handler.
3. In the created event handler write the following code:

```
if(Window.WndCaption == "Print")
{
```

```
Log.Message("Print window appeared, waiting...");
while(Window.Exists)
{
    aqUtils.Delay(500, "Print window appeared, waiting...");
}
}
```

4. Launch the `testUnexpectedWindowHandling1` function. As the function is working, press the following `Ctrl + P` keys combination, wait a little, and close the **Print** window.
5. Repeat the action several times. You will notice that script execution will be paused each time as the **Print** window shows up, and resumed after the **Print** window is closed.

### How it works...

The `OnUnexpectedWindow` event handler has the `Window` parameter, which is the targeted unexpected window, hindering work with `TestComplete`. This window could be handled as any other of its kind (refer to its properties and evoke its methods).

Via the `aqUtils.Delay` method, we are pausing the execution; otherwise, the `TestComplete` process will perform many checks in-between small intervals, which could slow down other running programs.

### See also

- ▶ The process of the event handler creation is thoroughly dealt with in the *Creating event handlers* recipe

## Handling unexpected windows that don't affect TestComplete

In some cases, it is necessary to capture appearance of a window which doesn't affect `TestComplete`, for example, a non-modal window that has been launched by another process. In this case, handling the `OnUnexpectedWindow` event will be of little use for us.

In this recipe, we will deal with an example of working with such a window. Our script will be working with the Notepad application, meanwhile closing the calculator window, if it has showed up.



## Getting ready

Launch the standard Windows Notepad (C:\Windows\notepad.exe) application and make sure that you can quickly launch Calculator Plus (for example, with the help of a shortcut or a given key's combination).

## How to do it...

To capture the Calculator Plus window appearing, it is necessary to perform the following actions:

1. First of all, we shall create a function that will execute the needed actions (that is, check if the calculator is up and running, and close it, if so):

```
function killCalc()
{
    if(Sys.WaitProcess("CalcPlus", 1, 1).Exists)
    {
        Log.Message("Calculator has been found");
        Sys.Process("CalcPlus").Terminate();
    }
}
```

2. Let's now create the function which will execute some of the timed actions: we will place a timer at its beginning with successive invocation of the previously created killCalc function:

```
function testUnexpectedWindowHandling2()
{
    Utils.Timers.Add(500, "Unit1.killCalc", true);
    var np = Sys.Process("NOTEPAD").Window("Notepad");
    np.Activate();
    for(var i = 1; i < 1000; i++)
    {
        np.Keys(i + "[Enter]");
        aqUtils.Delay(500);
    }
}
```

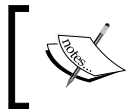
3. Launch the testUnexpectedWindowHandling2 function. During its execution time, launch the Calculator Plus once in a while.

You will see that each time the Calculator Plus is closed by TestComplete, the log is receiving the **Calculator has been found** message.

## How it works...

With the help of the `Utils.Timers.Add` method, we are creating a new timer, in line with the following parameters:

- ▶ The `interval` in terms of milliseconds within which the timer will fire up (in our case, 500 milliseconds)
- ▶ The name of the function which is necessary to launch as the timer fires up (in our case, it is the `Unit1.killCalc` function)



Please note that it is necessary to pass its complete name, that is, the name of the module and the name of the function, as dot-separated values.

- ▶ A `Boolean` to indicate if the timer should be on or off at its creation (in our case, this parameter is set to `true`, that is, the timer is created and immediately turned on)

Now, within the equal intervals, preset by the `Interval` parameter, any predefined and passed function will fire up. In our case, this function checks if the calculator has been launched, if so the function in view closes the calculator down.

## Saving the log to a disk after each test

One of the possible applications for capturing events is regular saving of the log to HDD to avoid the loss of results against unforeseen situations (blackouts, emergency `TestComplete` termination, and so on). This is particularly important in big projects, where there are many tests being executed and results being checked regularly.

In this recipe, we will consider how to store the log as each test completes.

## How to do it...



To save the results after each test execution is over, it is necessary to perform the following steps:

1. Create an event handle for the `OnStopTest` event.
2. Place the following code into the created handler:
 

```
Log.Event("Saving results to disk");
Log.SaveToDisk();
```
3. Create several testing functions with any contents, for example:



```
function Test1()
{
    Log.Message("Test 1");
}
```

4. Add these functions to test items (right-click on the name of the project, and navigate to **Edit | Test Items**). In the result, the listing of the tests will appear, as follows:

Name	Test
<input checked="" type="checkbox"/>  ProjectTestItem1	Script\Unit1 - Test1
<input checked="" type="checkbox"/>  ProjectTestItem2	Script\Unit1 - Test2 ...

5. Launch the whole project for execution by right-clicking on the name of the project and selecting the Run menu item.

In the result, a log will be instanced, thereby the results of each test execution will be saved to HDD (this is easy to verify by checking for the message at the end of each test):

Type	Message
	Test 2
	Saving results to disk

## How it works...

Saving the log to HDD is possible at any given moment of time by invoking the `Log.SaveToDisk` method.

We are doing so at the end of each test, using the `OnStopTest` event. Pay attention that the event is triggered at the end of each test and launched via test items. If you simply launch the function from which you call several tests, the event will fire just once (as each invoked function completes its call), rather than upon each invoked test. For example:

```
function TestAll()
{
    Test1();
    Test2();
    // OnStopTest
}
```

## See also

- ▶ More details about adding event handlers can be read about in the *Creating event handlers* recipe
- ▶ Creation of test items is thoroughly dealt with in the *Creating a test plan for regular runs* recipe in *Chapter 4, Running Tests*

## Sending a notification e-mail on timeouts

In this recipe we will deal with an example of dispatching an e-mail in case the execution time of any given test exceeds the maximally assigned time frame.

### How to do it...

To send an e-mail upon reaching the timeout, it is necessary to follow through these actions:

1. Create an `OnTimeout` event handler.
2. Add the following code into the created handler:
 

```
BuiltIn.SendMail("recipient@example.com", "example.com",
  "Sender Name", "sender@example.com", "Subject", "Body");
```
3. Create a function that will work for 62 seconds. For example, as follows:
 

```
function testEmailOnTimeout()
{
  aqUtils.Delay(62*1000);
}
```
4. Create the test item (right-click on the name of the project and navigate to **Edit | Test Items**) and set the timeout to be equal to 1 minute as shown in the following screenshot:

Name	Test	Timeout...	C
<input type="checkbox"/> Proj...	Script\Unit1 - Test1	0	
<input type="checkbox"/> Proj...	Script\Unit1 - Test2	0	
<input checked="" type="checkbox"/> Test3	Script\Unit1 - testEmailOnTimeout	1	

5. Launch the project for execution (right-click on the name of the project and select the **Run** menu).

In the result, as the one-minute timespan goes out, the timeout will trigger the e-mail dispatch.

### How it works...

If a timeout has been signified for the test item, as it fires, the corresponding event will be triggered. In our case, as the timeout fires, the e-mail will be sent to the following address: `recipient@example.com`.

We have used 62-seconds timeout because the minimum timeout is 1 minute and our example should work for more than 1 minute. In such cases it's usually better to have some reserve (for example, 2 seconds, as we have here) to avoid casual firing of the events.

It is also a good practice to specify timeouts by multiplying the number of seconds directly in the code (as we have done here), because such constructions are more maintainable in the future.

### See also

- ▶ The considered example of dispatching an e-mail does not suit all of the instances. A more universal method is considered in the *Sending logs via e-mail* recipe in *Chapter 6, Logging Capabilities*.
- ▶ The process of creation of the event handler is thoroughly dealt with in the *Creating event handlers* recipe.
- ▶ Test Items creation is described in greater detail in the *Creating a test plan for regular runs* recipe in *Chapter 4, Running Tests*.

## Creating preconditions and postconditions for tests

Preconditions and postconditions are often used both in manual and in automated tests for the purpose of assigning testing conditions or for returning the settings previously changed in the process of the test execution.

One of the most frequently used examples of pre-and postconditions for automated tests is launching and closing of the testing application. Such an approach explicitly allows us to avoid having to repeat these actions at the beginning of the test.

In this recipe we will consider an example of handling the Notepad application, which will automatically open and close in the pre-and postconditions, respectively.

### How to do it...

To create the tests with pre- and postconditions, it is necessary to carry out the following actions:

1. Create the `OnStartTest` event handler and add the following code to it:

```
Log.Event("Starting Notepad");  
Sys.OleObject("WScript.Shell").Run("notepad.exe");
```



2. Create the `OnStopTest` event handler and add the following code to it:

```
Log.Event("Stopping Notepad");
Sys.Process("notepad").Terminate();
```

3. Now, let's create several test examples:

```
function testNotepad1()
{
    var np = Sys.Process("NOTEPAD").Window("Notepad");
    np.Keys("Test #1");
}
function testNotepad2()
{
    var np = Sys.Process("NOTEPAD").Window("Notepad");
    np.Keys("Test #2");
}
```

4. Let's create the corresponding test items (right-click on the name of the project and navigate to **Edit | Test Items**). This should appear as follows in the end-result:

Name	Test
<input checked="" type="checkbox"/>  TestNotepad1	Script\Unit1 - testNotepad1
<input checked="" type="checkbox"/>  TestNotepad2	Script\Unit1 - testNotepad2

5. Let's launch the project (right-click on the name of the project and select the **Run** menu item).

In the result, all the tests will be launched, while each one of them will have a new copy of the notepad opened. As the test completes, the Notepad application will be closed.

### How it works...

The events `OnStartTest` and `OnStopTest` will be processed accordingly, before and after the tests are launched.

These events will be triggered individually per each test only in case the tests are launched via test items. If several tests should be launched in series from one and the same function, the events will be triggered just once (before the main function invocation and on its completion).

## See also

- ▶ Reading up on adding event handlers is available in the *Creating event handlers* recipe
- ▶ Launching and completion of programs is particularly dealt with in the *Running external programs and DOS commands*, *Running a tested application from the script*, and *Terminating a tested application* recipes in *Chapter 2, Working with Tested Applications*

# Index

## Symbols

**/exit parameter 112**  
**/ns parameter 106**  
**/project parameter 104**  
**/quiet parameter 53**  
**/routine parameter 104**  
**/run parameter 104**  
**/SilentMode parameter 176**  
**(UnitA.calledFunction()) function 90**  
**/Unit parameter 104**

## A

**Access Database Engine. *See* ACE**  
**AccessibilityObject property 81**  
**ACE 216**  
**Action after run parameter 234**  
**actions**  
    adding, to Keyword Tests 188-190  
**Activate button 10**  
**Activate license button 10**  
**Add button 144**  
**additional parameters**  
    passed, for testing 106-108  
**AdditionalText property 247**  
**Add Variable button 191**  
**Append to Test button 190**  
**aqObject.CheckProperty method 67, 68**  
**aqObject.CompareProperty function 68**  
**aqObject.CompareProperty method 212**  
**aqObject.IsSupported method 133, 134**  
**aqObject object 132**  
**aqUtils.Delay method 24, 116, 138, 249**

**attrBoldBlue variable 160**  
**Automatic activation option 10**  
**automatic runs**  
    scheduling, at nighttime 110-112

## B

**BackColor parameter 160**  
**batch-replacement symbols**  
    \* (asterisk) 60  
    ? (question mark) 60  
**Bold parameter 160**  
**Bounds.Left property 81**  
**Bounds property 81**  
**Breakpoint Properties window 178**  
**breakpoints**  
    used, for script execution pausing 177-179  
**Browsers object 223**  
**btn2 variable 17, 18**  
**btn6 button 187, 188**  
**BuiltIn object 184**  
**BuiltIn.ParamCount method 106**  
**BuiltIn.ParamStr method 106**  
**BuiltIn.ShowMessage method 119**

## C

**CalcPlus element 37, 193**  
**calledFunction function 90**  
**callee method 132**  
**callerFunction function 90, 91**  
**call function 91**  
**CallObjectMethodAsync method 132**  
**Cancel button 16**  
**catch block 88**



- CDO** 171
- CE button** 188
- checkCalcResult function** 211
- checkContent method** 92, 94
- CHECK keyword** 200
- checkNotepadFileName function** 200
- Checkpoints element** 194
- CheckProperty method** 67
- Clear method** 118
- ClickButton element** 189
- clickCalcButton function** 211
- clickDisabledObject function** 247
- Click method** 131
- ClientX parameter** 188
- ClientY parameter** 188
- CloseDriver method** 207
- CLOSE keyword** 200
- Close method** 44, 45, 92, 93, 125
- closeNotepad function** 200
- code**
  - structuring, loops used 61-63
- code templates**
  - creating 22, 23
- Collaboration Data Objects.** *See* **CDO**
- command line**
  - additional parameters passed, for testing 106-108
  - tests, running from 103-106
- CommandLine property** 107
- CommonOperations unit** 85
- Common unit** 85
- Compare method** 157
- CompareProperty method** 68
- Computer element** 47
- Condition drop-down** 66
- condition parameter** 67
- configuration files**
  - using, for workstations 236, 237
- Configure button** 167
- contentText property** 225
- Continue Execution command** 182
- ConversionOperations unit** 85
- Convert to Script menu item** 198
- CopyProjectToSlave method** 235
- Copy remote log parameter** 234
- counter variable** 191, 193
- Count parameter** 38

- Create button** 12
- Create Object Checkpoint window** 194
- Create Project Item dialog window** 26
- Create Project Item window** 16, 242
- Create Property Checkpoint option** 64
- cross-browser testing**
  - performing 222-224
- CSV delimiter**
  - changing 212, 213
- CurrentDriver property** 215
- currentRow variable** 207
- custom control classes**
  - mapping, to standard classes 140-143
- customizable settings**
  - restoring 20, 21
  - saving 20, 21

**D**

- data**
  - generating, Data Generator used 204, 205
  - reading, from table 208, 209
  - sharing, between workstations 237-239
- Data Connectivity Components** 215
- data-driven approach** 79
- Data-driven Testing.** *See* **DDT**
- Data Generator**
  - used, for data generating 204, 205
- Data Generator Wizard window** 204
- DDT**
  - about 203, 237
  - Excel spreadsheets, handling 215
  - used, for accessing cell in Excel 206, 207
- DDT.ExcelDriver method** 208, 215, 216
- DDTExcel function** 216
- DDT tables**
  - handling 214
  - used, for expected values storing 209-211
- debugging**
  - about 175
  - disabling 176
  - enabling 176
  - step-by-step instructions 181, 182
- Debug mode**
  - tested application, running in 45, 46
- Delay method** 119
- Delete operation** 26

- DelphiScript language 14**
- Depth parameter 136**
- description column 187**
- Digit grouping option 136**
- disabled controls**
  - clicking 245-247
- disableSleep function 117**
- disk**
  - log, saving to 251, 252
- Display Object Spy button 32**
- distributed testing**
  - about 232
  - Master computer 233
  - Network Suite, configuring for 232-234
  - Slave computer 233
- document.all element 219**
- document.frames object 219**
- DOS commands**
  - running 50-52
- dotNET object 14**
- Drag method 127, 130**
- DriveMethod method 214**
- dynamic content**
  - screenshots, comparing with 165-167

## **E**

- Edit button 204**
- Edit unit 85**
- element appearance**
  - awaiting, on webpage 225-227
- e-mail**
  - log, sending via 171-173
- Enabled property 135**
- enterStr variable 58**
- error count**
  - obtaining, in log 161-163
- error messages post**
  - disabling 244, 245
- ErrorsMax variable 162**
- ErrorsTotal variable 161**
- eval function 247**
- Evaluate button 183**
- Evaluate window**
  - about 183
  - working 183, 184

- events action 168**
- Events element 161**
- events handlers**
  - creating 242-244
- Excel cell**
  - accessing, DDT used 206, 207
- Excel driver**
  - auto-detecting 216
- ExcelDriver method 216**
- Excel spreadsheets**
  - accessing, via DDT 215
- exceptions**
  - handling 86-88
  - handling, from different unit 89-91
- Exists method 227**
- Exists property 138, 227**
- expected values**
  - storing, DDT tables used 209-211
- Export button 20**
- expressions**
  - evaluating 182-184
- external programs**
  - running 50-52

## **F**

- fileName parameter 200**
- finally block 88**
- FindAllChildren method 134, 136**
- FindAll method 136**
- FindChild method 136**
- Find method 134, 135, 157**
- FindRectByText method 147**
- Find window 126, 127**
- Finish button 66**
- FirstName property 208**
- fn1 function 182**
- folders**
  - creating, in log 158, 159
- FontColor parameter 160**
- For Loop window 192**
- framework**
  - creating, OOP approach used 92-95
- function**
  - running 98, 99
- functional decomposition approach 83**

## G

**GeneralEvents\_OnLogError function** 243

**global variables**

using 73-75

**grid.ScreenTop property** 81

**GroupIndex parameter** 28

## H

**hard disk drive (HDD)** 12

**Help unit** 85

**Hide method** 118

**Highlight Object button** 34

## I

**image format action** 168

**Import button** 21

**Index parameter** 223

**Indicator**

messages, sending to 118, 119

**Indicator object** 118

**Insert button** 204

**Inspect button** 183

**installers**

testing 52, 53

**Interval parameter** 251

**IsSupported method** 132

**isTextPresent function** 225

**item column** 187

**Item property** 80, 223

**Items property** 31

## J

**jobs**

about 234

parameters 234

**join method** 91

**JScript language** 14

## K

**Keys method** 45, 57, 58

**Keyword-driven Testing** 186

**Keyword driver**

components 200

creating 199-202

**Keyword Tests**

about 185

actions, adding to 188-190

converting, to scripts 198, 199

enhancing, loops used 191-193

operations, representing 187

recording 186-188

script functions, calling from 195-197

**Keyword Tests element** 188

**killCalc function** 250

## L

**Language combobox** 13

**latest browsers versions**

updates, using 221

**level of application's common code layer** 84

**level of libraries** 84

**level of tests** 84

**LinkCount parameter** 141

**linkLogin element** 227

**linkLoginLink element** 226

**Link parameter** 141

**Load From File button** 23

**LoadFromFile method** 157

**Locked parameter** 245

**log**

error count, obtaining 161-163

exporting, in MHT format 170, 171

folders, creating in 158, 159

generating, in text format 168-170

message appearance, changing 159, 160

messages, posting to 154, 155

saving, to disk after each test 251, 252

screenshots, posting to 156, 157

sending, via e-mail 171-173

**Log.CreateFolder method** 158, 159

**Log.CreateNewAttributes method** 160

**Log.ErrCount property** 161

**Log.Error method** 244

**Log.Event method** 246

**Login button** 9

**Log.Message** method 29, 58, 98, 160, 177  
**LogParams** object 245, 247  
**LogParams.Str** method 169  
**Log.Picture** method 147, 156  
**Log.PopLogFolder** method 158  
**Log.PushLogFolder** method 158, 159  
**Log.SaveResultsAs** method 171  
**Log.SaveToDisk** method 252  
**log size**  
decreasing 167, 168  
**log size, decreasing**  
events generation, disabling 167  
image format, changing 167  
Visualizer, disabling 167  
**loops**  
used, for code structuring 61-63  
used, for Keyword Tests enhancing 191-193

## M

**main function** 181  
**MainMenu** property 31  
**Manual Tests** element  
adding, to project 25-27  
**Master workstation**  
used, for running tests 235  
**messages**  
Log.Error message 155  
Log.Event message 155  
Log.File message 155  
Log.Link message 155  
Log.Message message 155  
Log.Warning message 155  
posting, to logs 154, 155  
sending, to Indicator 118, 119  
**MessageText** parameter 245  
**MessageText** property 245  
**MessageType** parameter 68  
**message window**  
showing, during script run 119, 120  
**methods**  
asynchronously calling 131, 132  
**MHT** format  
log, exporting in 170  
**MSI** file  
running 52, 53

**multilingual applications**  
testing 75-79

## N

**NameMapping** element 16, 78, 79  
**NameMapping** object 79  
**NameMapping** window 123  
**NavigateTo** method 229  
**Network Suite**  
about 231  
configuring, for distributed testing 232-234  
**NetworkSuite** element 239  
**NetworkSuite.Synchronize** method 240  
**NetworkSuite.Variables.Var1** variable 238  
**NetworkSuite.WaitForState** method 236  
**Network variables**  
used, for data sharing between workstations  
237-239  
**New Event Handler** window 243  
**New Test Item** button 102  
**Next** method 207  
**nonstandard controls**  
working with 79-83  
**nonstandard controls text**  
accessing, text recognition used 143-145  
**notepad** variable 59  
**notification e-mail**  
sending, on timeouts 253, 254

## O

**object**  
appearance, awaiting 136-138  
dragging, onto object 127-130  
finding, property values used 134, 135  
**Object Browser** panel  
about 29  
working 30-32  
**Object Browser** tab 30  
**Object Browser** tree 32  
**Object Checkpoint**  
creating, in Keyword Tests 193-195  
**object checkpoints**  
creating 68-72  
**Object Mapping** 140  
**Object-Oriented Programming**. *See* OOP

**object parameter** 67  
**Objects element** 71  
**Object Spy utility**  
about 28  
using 32-34  
**Object Spy window** 33  
**Object Tree Model**  
preferring 122-124  
**Object tree model parameter** 122, 123  
**OCR**  
using 145-149  
**OCROptions.ExactSearch property** 148  
**OCROptions.SearchAccuracy property** 148  
**ODBC** 216  
**ODT (Object-Driven Testing) object** 95  
**OnLogError element** 243  
**OnLogError event** 161, 242, 246  
**On-Screen Action element** 189  
**On-Screen Action window** 187  
**OnStartTest event** 254  
**OnStopTest event** 251, 252, 255  
**OnTimeout event** 253  
**OnUnexpectedWindow event** 248, 249  
**OOP**  
about 92  
used, for framework creating 92-95  
**Open Database Connectivity.** *See* ODBC  
**openFile method** 92, 94  
**OPEN keyword** 200  
**operation column** 187  
**Operation Parameters window** 196  
**Optical Character Recognition.** *See* OCR  
**OSInfo.FullName property** 237  
**OSInfo.Name parameter** 237  
**overlapping windows**  
ignoring 125-127

## P

**PagePicture method** 228  
**Params button** 31  
**pCalc variable** 28  
**people variable** 209  
**persistent variable** 74  
**pEx variable** 183

**picture format**  
BMP 164  
changing 163-165  
JPEG 164  
PNG 164  
TIFF 164  
**Picture method** 80, 150, 157, 228  
**picWindow variable** 156  
**Playback element** 114  
**playback options**  
Auto-wait timeout 114  
changing 113-115  
Delay between events 114  
Disable mouse 114  
Dragging delay 114  
Ignore overlapping window 115  
Key pressing delay 114  
Minimize TestComplete 114  
Mouse movement delay 114  
On unexpected window 114  
Post image on error 115  
Save log every N minutes 115  
Stop on error 114  
Stop on warning 114  
Stop on window recognition error 114  
Store last N events 115  
**pNotepad variable** 180  
**PopText method** 119  
**postconditions**  
creating, for tests 254, 255  
**preconditions**  
creating, for tests 254, 255  
**printPersonLastName function** 214  
**Process method** 28  
**procName parameter** 42  
**project**  
copying, to Slave workstation 234  
creating 11, 12  
script code, organizing in 83-86  
scripting language, selecting 13, 14  
**Project Explorer panel** 11  
**project items**  
adding 25-27  
removing 25-27  
**Project.TestItems object** 103  
**Properties... button** 70

**property checkpoint**  
creating 64-68  
**PropertyName parameter 140**  
**property parameter 67**  
**PropertyValue parameter 140**  
**property values**  
awaiting 139, 140  
used. for objects finding 134-136  
**PushText method 118**

## R

**recorded test**  
modifying 18-20  
**Recording pane 64**  
**Recording panel 69, 71**  
**Rect object 151**  
**Refresh method 135**  
**Regions.AddPicture method 151, 166**  
**Regions.Compare method 166, 167**  
**Regions element 151**  
**Regions.FindRegion method 150**  
**regular runs**  
test plans, creating for 101-103  
**Remote Desktop**  
tests, running via 112, 113  
**Remove operation 26**  
**res1 variable 182**  
**RunAll method 40**  
**RunAs method 49**  
**Run method 39, 52, 236**  
**Run Mode property 39**  
**Runner.CallObjectMethodAsync method 132**  
**Runner.Pause method 179**  
**Runner.Stop method 162**  
**Run Script Routine element 196**  
**run speed**  
increasing 115, 116  
**Run to Cursor command 182**

## S

**Save to File button 23**  
**SaveToFile method 156, 157**  
**Save to User-defined button 205**  
**ScientificOperations unit 85**

**screensaver**  
disabling, during script execution 116, 117  
**screenshots**  
comparing, with dynamic content 165-167  
posting, to logs 156, 157  
**script code**  
organizing, in project 83-86  
**Script element 102, 163, 197**  
**script execution**  
delaying 24, 25  
pausing, breakpoints used 177-179  
screensaver, disabling during 116, 117  
**script functions**  
calling, Keyword Tests used 195-197  
**scripting 55**  
**script run**  
message window, showing during 119, 120  
**scripts**  
Keyword Tests, converting to 198, 199  
running, on webpage 229, 230  
tested application, running from 38-40  
tested application settings, dynamically  
changing from 49, 50  
**Scripts element 84**  
**Select button 145**  
**Select Features window 9**  
**Select Test window 196**  
**SendEmail function 172**  
**Send method 172**  
**set command 112**  
**SetThreadExecutionState function 117**  
**Settings button 146**  
**Show method 118**  
**Sign Up Now button 8**  
**Size.Height property 157**  
**Size.Width property 157**  
**Slave workstation**  
project, copying to 234  
**slPacker object 172**  
**slPacker.PackCurrentTest method 173**  
**specific object property**  
verifying 132-134  
**Specify Routine Name window 198**  
**split method 91, 247**  
**sqrt button 146, 147**  
**standard classes**  
custom control classes, mapping to 140-143

- start method** 92, 93
- startNotepad function** 200
- Step Into command** 182
- Step Over command** 182
- Stop button** 16, 66
- Stores element** 71, 72, 149
- Stretch method** 156, 157
- Str property** 245
- substr method** 211
- SynchPoints element** 239
- Sys.Desktop.MouseDown method** 247
- Sys.Desktop.MouseUp method** 247
- Sys element** 142
- Sys.HostName property** 237
- SysLink class** 142
- SysLink element** 142
- Sys object** 30
- Sys.OSInfo.Windows64bit property** 50
- SystemParametersInfo function** 116

## T

### table

- data, reading from 208, 209

- taskkill command** 111

- Task Scheduler window** 111

- TCPATH variable** 112

- Terminate method** 41, 42

### test

- debugging, instructions 181, 182
- postconditions, creating 254-256
- preconditions, creating 254-256
- recording 15-17
- recording, prerequisite steps 15
- recording, steps 15-17
- running, from command line 103-106
- running, Master workstation used 235
- running, via Remote Desktop 112, 113

- Test1 function** 17, 223

- Test1Modified2 function** 24

- Test1Modified3 function** 20

- Test2 function** 223

### test accuracy

- verifying 100, 101

- Test Actions element** 196

- testClickDisabledButton function** 246

### TestComplete

- about 7
- batch-replacement symbols 60
- customizable settings 20
- installing 8-11
- interacting, with tested applications 27-29
- license types 10
- logging capabilities 153
- Network Suite feature 231
- project, creating 11, 12
- system requirements 8
- tested application project item,
  - adding to 36-38
- unexpected windows, handling 248-251
- unsupported controls, using 149-151
- variables 74

### TestComplete license types

- floating license 10
- node-locked license 10

### TestComplete parameter 107

- testCSVDelimiter function** 213

- testDriveMethod function** 214

### tested application

- closing 43-45
- instances, killing 41, 42
- running, from script 38-40
- running, in Debug mode 45, 46
- running, under different user account 47-49
- settings, dynamically changing from
  - script 49, 50
- terminating 40, 41
- TestComplete, interacting with 27-29

### tested application project item

- adding, to TestComplete 36-38

- TestedApps.CalcPlus.Run() command** 48

- TestedApps element** 36, 37, 47, 92

- TestedApps.notepad.Run() function** 45

- testEditControl1 function** 56

- testEditControl2 function** 56

- testErrorsCount function** 162

- testEvaluate function** 183

- testExistingVars function** 74

- testExpectedValues function** 210, 211

- Test Items project element** 101

- testNewVars function** 74

**test plan**  
  creating, for regular runs 101-103  
**test plan runs**  
  organizing 109, 110  
**test runs**  
  synchronizing, on workstations 239, 240  
**testUnexpectedWindowHandling1**  
  function 249  
**testUnexpectedWindowHandling2**  
  function 250  
**testVariables** function 74  
**testVerifyText** function 225  
**text**  
  entering, into text fields 56-58  
**text availability**  
  verifying, on webpage 224, 225  
**text fields**  
  text, entering into 56-58  
**Text** property 118  
**text recognition method**  
  used, for nonstandard controls text  
    accessing 143-145  
**Timeout** parameter 140  
**timeouts**  
  notification e-mail, sending on 253, 254  
**toArray** method 134, 136  
**Tolerance** parameter 167  
**toString** method 91  
**ToUrl** method 229  
**Transparent** parameter 166  
**try** block 88

## U

**Unexpected window** 127  
**unexpected windows, handling**  
  TestComplete, affecting 248, 249  
  TestComplete, not affecting 249-251  
**Unit1.killCalc** function 251  
**Update** objects option 72  
**URL** parameter 229  
**UseACEDriver** parameter 216  
**Use previous instance** parameter 234  
**User Forms** 120

**Utils.Picture** property 157  
**Utils.Timers.Add** method 117, 251

## V

**value column** 187  
**value parameter** 67  
**Var1** variable 237, 239  
**variable values**  
  viewing 179, 180  
**varying name objects**  
  processing, wildcards used 58-61  
**VBAArray** object 136  
**VBScript** language 14  
**VCLObj** method 29  
**visualizer** action 168

## W

**WaitAliasChild** method 227  
**WaitForCompletion** method 132  
**WaitForCompletion** parameter 236  
**WaitForNetVarChange** method 239  
**Wait** method 24, 225  
**WaitNamedChild** method 227  
**WaitProcess** method 42, 43  
**WaitProperty** method 139, 140  
**WaitQtObject** method 138  
**WaitWindow** method 44, 63, 136  
**WaitWinFormsObject** method 138  
**wCalc** variable 28, 125  
**web applications**  
  models 219, 220  
  testing 217  
**web applications models**  
  DOM 219  
  Hybrid 220  
  Tag 220  
  Tree 219  
**webpage**  
  element appearance, awaiting on 225-227  
  screenshot, creating 228  
  scripts, running on 229, 230  
  text availability, verifying on 224, 225  
**Web Tree Model**  
  settings, changing 218-221



**wildcards**

used, for varying name objects processing  
58-61

**Window class 60****Window method 28****Window parameter 249****windows life cycle**

about 124, 125

**WinFormsObject method 29****wMain method 92, 93****WndCaption parameter 28****WndCaption property 31, 77****WndClass parameter 28****WndClass property 28****wndSciCalc element 71****wndSciCalc object 71****wndSciCalc variable 17, 18****wOpen method 92, 93****workstations**

data, sharing between 237-239

different configuration files, using  
for 236, 237

test runs, synchronizing on 239, 240

**WScript.Shell object 50-52****wText property 28, 57, 133, 141****X****X parameter 128****Y****Y parameter 128**

**[PACKT]** Thank you for buying  
PUBLISHING **TestComplete Cookbook**

## About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

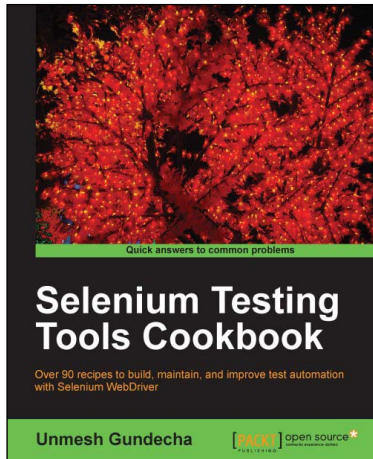
Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: [www.packtpub.com](http://www.packtpub.com).

## Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to [author@packtpub.com](mailto:author@packtpub.com). If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

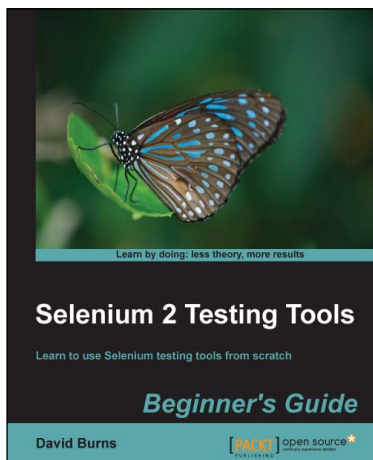


## Selenium Testing Tools Cookbook

ISBN: 978-1-84951-574-0      Paperback: 326 pages

Over 90 recipes to build, maintain, and improve test automation with Selenium WebDriver

1. Learn to leverage the power of Selenium WebDriver with simple examples that illustrate real world problems and their workarounds
2. Each sample demonstrates key concepts allowing you to advance your knowledge of Selenium WebDriver in a practical and incremental way
3. Explains testing of mobile web applications with Selenium Drivers for platforms such as iOS and Android



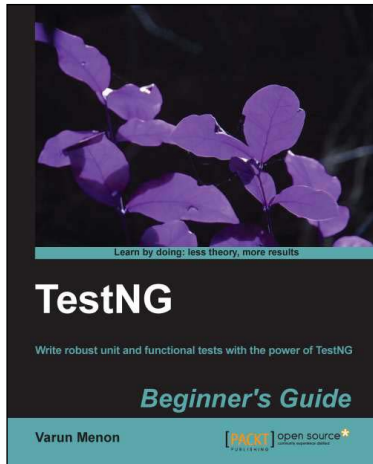
## Selenium 2 Testing Tools Beginner's Guide

ISBN: 978-1-84951-830-7      Paperback: 232 pages

Learn to use Selenium testing tools from scratch

1. Automate web browsers with Selenium WebDriver to test web applications
2. Set up Java Environment for using Selenium WebDriver
3. Learn good design patterns for testing web applications

Please check [www.PacktPub.com](http://www.PacktPub.com) for information on our titles

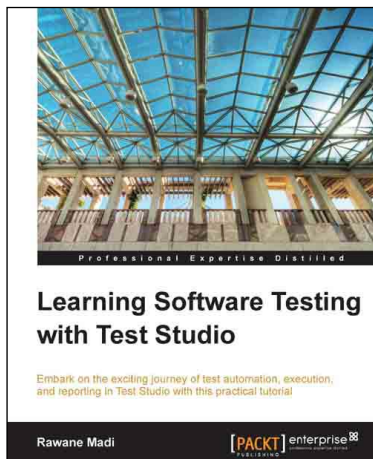


## TestNG Beginner's Guide

ISBN: 978-1-78216-600-9      Paperback: 276 pages

Write robust unit and functional tests with the power of TestNG

1. Step-by-step guide to learn and practise any given feature
2. Detailed understanding of the features and core concepts
3. Learn about writing custom reporting



## Learning Software Testing with Test Studio

ISBN: 978-1-84968-890-1      Paperback: 376 pages

Embark on the exciting journey of test automation, execution, and reporting in Test Studio with this practical tutorial

1. Learn to use Test Studio to design and automate tests valued with their functionality and maintainability
2. Run manual and automated test suites and view reports on them
3. Filled with practical examples, snapshots and Test Studio hints to automate and substitute throwaway tests with long term frameworks

Please check [www.PacktPub.com](http://www.PacktPub.com) for information on our titles