

**Effective GUI  
Test Automation:  
Developing an Automated  
GUI Testing Tool**

*Kanglin Li  
Mengqi Wu*

**SYBEX®**

# Effective GUI Test Automation: Developing an Automated GUI Testing Tool

Kanglin Li and Mengqi Wu



SYBEX

San Francisco · London

Associate Publisher: Joel Fugazzotto  
Acquisitions and Developmental Editor: Tom Cirtin  
Production Editor: Erica Yee  
Technical Editor: Acey J. Bunch  
Copyeditor: Judy Flynn  
Compositor: Laurie Stewart, Happenstance Type-O-Rama  
Graphic Illustrator: Jeff Wilson, Happenstance Type-O-Rama  
Proofreaders: Laurie O'Connell, Amy Rassmussen, Nancy Riddiough  
Indexer: Ted Laux  
Cover Designer: Ingalls + Associates  
Cover Illustrator/Photographer: Rob Atkins, Image Bank

Copyright © 2005 SYBEX Inc., 1151 Marina Village Parkway, Alameda, CA 94501. World rights reserved. The author(s) created reusable code in this publication expressly for reuse by readers. Sybex grants readers limited permission to reuse the code found in this publication or its accompanying CD-ROM so long as the author(s) are attributed in any application containing the reusable code and the code itself is never distributed, posted online by electronic transmission, sold, or commercially exploited as a stand-alone product. Aside from this specific exception concerning reusable code, no part of this publication may be stored in a retrieval system, transmitted, or reproduced in any way, including but not limited to photocopy, photograph, magnetic, or other record, without the prior agreement and written permission of the publisher.

Library of Congress Card Number: 2004108202

ISBN: 0-7821-4351-2

SYBEX and the SYBEX logo are either registered trademarks or trademarks of SYBEX Inc. in the United States and/or other countries.

TRADEMARKS: SYBEX has attempted throughout this book to distinguish proprietary trademarks from descriptive terms by following the capitalization style used by the manufacturer.

The author and publisher have made their best efforts to prepare this book, and the content is based upon final release software whenever possible. Portions of the manuscript may be based upon pre-release versions supplied by software manufacturer(s). The author and the publisher make no representation or warranties of any kind with regard to the completeness or accuracy of the contents herein and accept no liability of any kind including but not limited to performance, merchantability, fitness for any particular purpose, or any losses or damages of any kind caused or alleged to be caused directly or indirectly from this book.

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

## **SOFTWARE LICENSE AGREEMENT: TERMS AND CONDITIONS**

The media and/or any online materials accompanying this book that are available now or in the future contain programs and/or text files (the “Software”) to be used in connection with the book. SYBEX hereby grants to you a license to use the Software, subject to the terms that follow. Your purchase, acceptance, or use of the Software will constitute your acceptance of such terms.

The Software compilation is the property of SYBEX unless otherwise indicated and is protected by copyright to SYBEX or other copyright owner(s) as indicated in the media files (the “Owner(s)”). You are hereby granted a single-user license to use the Software for your personal, noncommercial use only. You may not reproduce, sell, distribute, publish, circulate, or commercially exploit the Software, or any portion thereof, without the written consent of SYBEX and the specific copyright owner(s) of any component software included on this media.

In the event that the Software or components include specific license requirements or end-user agreements, statements of condition, disclaimers, limitations or warranties (“End-User License”), those End-User Licenses supersede the terms and conditions herein as to that particular Software component. Your purchase, acceptance, or use of the Software will constitute your acceptance of such End-User Licenses.

By purchase, use or acceptance of the Software you further agree to comply with all export laws and regulations of the United States as such laws and regulations may exist from time to time.

### **Reusable Code in This Book**

The author(s) created reusable code in this publication expressly for reuse by readers. Sybex grants readers limited permission to reuse the code found in this publication, its accompanying CD-ROM or available for download from our website so long as the author(s) are attributed in any application containing the reusable code and the code itself is never distributed, posted online by electronic transmission, sold, or commercially exploited as a stand-alone product.

### **Software Support**

Components of the supplemental Software and any offers associated with them may be supported by the specific Owner(s) of that material, but they are not supported by SYBEX. Information regarding any available support may be obtained from the Owner(s) using the information provided in the appropriate read.me files or listed elsewhere on the media.

Should the manufacturer(s) or other Owner(s) cease to offer support or decline to honor any offer, SYBEX bears no responsibility. This notice concerning support for the Software is provided for your information only. SYBEX is not the agent or principal of the Owner(s), and SYBEX is in no way responsible for providing any support for the Software, nor is it liable or responsible for any support provided, or not provided, by the Owner(s).

### **Warranty**

SYBEX warrants the enclosed media to be free of physical defects for a period of ninety (90) days after purchase. The Software is not available from SYBEX in any other form or media than that enclosed herein or posted to [www.sybex.com](http://www.sybex.com). If you discover a defect in the media during this warranty period, you may obtain a replacement of identical format at no charge by sending the defective media, postage prepaid, with proof of purchase to:

SYBEX Inc.  
Product Support Department  
1151 Marina Village Parkway  
Alameda, CA 94501  
Web: <http://www.sybex.com>

After the 90-day period, you can obtain replacement media of identical format by sending us the defective disk, proof of purchase, and a check or money order for \$10, payable to SYBEX.

### **Disclaimer**

SYBEX makes no warranty or representation, either expressed or implied, with respect to the Software or its contents, quality, performance, merchantability, or fitness for a particular purpose. In no event will SYBEX, its distributors, or dealers be liable to you or any other party for direct, indirect, special, incidental, consequential, or other damages arising out of the use of or inability to use the Software or its contents even if advised of the possibility of such damage. In the event that the Software includes an online update feature, SYBEX further disclaims any obligation to provide this feature for any specific duration other than the initial posting.

The exclusion of implied warranties is not permitted by some states. Therefore, the above exclusion may not apply to you. This warranty provides you with specific legal rights; there may be other rights that you may have that vary from state to state. The pricing of the book with the Software by SYBEX reflects the allocation of risk and limitations on liability contained in this agreement of Terms and Conditions.

### **Shareware Distribution**

This Software may contain various programs that are distributed as shareware. Copyright laws apply to both shareware and ordinary commercial software, and the copyright Owner(s) retains all rights. If you try a shareware program and continue using it, you are expected to register it. Individual programs differ on details of trial periods, registration, and payment. Please observe the requirements stated in appropriate files.

### **Copy Protection**

The Software in whole or in part may or may not be copy-protected or encrypted. However, in all cases, reselling or redistributing these files without authorization is expressly forbidden except as specifically provided for by the Owner(s) therein.

*To Li Xuanhua and Tang Xunwei*

*And*

*In memory of Luo Youai, Luo Xulin,  
and Li Congyang*

# Acknowledgments

---

**T**o the folks at Sybex, especially Tom Cirtin who made this book available, Acey J. Bunch for his technical expertise, and Erica Yee and Judy Flynn for their contributions to this book. I want to thank Rodney Zaks, the president of Sybex, who signed the contract. I also extend my thanks to the other people in the Sybex team.

I still need to thank other people, especially the readers who have provided me with their comments and suggestions about my previous works. During the process of writing this book, the test monkey in Chapter 3 frequently reminded me of my early days as a schoolboy. My friends and I spent most of my childhood summer nights listening to the stories of the monkey king told by the old folks in our village. What I learned from both the old folks and my friends makes it possible for me to write this book today.

When I was a small boy, my uncle OuYang Minghong carried me in a bamboo basket to visit my grandma. My uncles Zuomei and Zuodian carried me on their backs or in a wheelbarrow to watch movies and catch fish. Since I was five, my aunt Youai led me to the school and taught me to count with my fingers.

I didn't learn how to swim and fell into a pool when I was eight or nine. Meiqing risked his life and saved me from the water with the help of my uncle Zuohuang, Guicheng, and another whose name I don't know. Meiqing has always been my big brother. Then I learned how to swim at 20 and swam across the Xiangjiang River on a summer day and the Leishui River on a cold fall afternoon. Thank you all for teaching me about water, including Zeng Xilong, Li Wenbing, Kuang Chuanglun, Bai Maoye, Chen Xiaohu, Zeng Yigui, He Hechu, Wen Xiaoping, Long Yongcheng, and Xie Hengbing.

Tang Duzhang, Li Zuojin, Li Zuojun, Luo Xixing, Chen Xinhua, and Kuang Chuangren and I spent our best years in the middle school. At that time we learned food was valuable. Together, we discovered that boys were shy and girls were caring. Thanks to all of the former classmates who form the memory of my childhood.

—Kanglin Li  
Leiyang, China  
2004

# Contents at a Glance

---

<i>Introduction</i>	<i>xiv</i>
<b>Chapter 1:</b> GUI Testing: An Overview	<b>1</b>
<b>Chapter 2:</b> Available GUI Testing Tools vs. the Proposed Tool	<b>19</b>
<b>Chapter 3:</b> C# Win32 API Programming and Test Monkeys	<b>37</b>
<b>Chapter 4:</b> Developing a GUI Test Library	<b>99</b>
<b>Chapter 5:</b> .NET Programming and GUI Testing	<b>137</b>
<b>Chapter 6:</b> Testing a Windows Form in General	<b>175</b>
<b>Chapter 7:</b> Architecture and Implementation of the Automatic GUI Test Tool	<b>209</b>
<b>Chapter 8:</b> Methods of GUI Test Verification	<b>269</b>
<b>Chapter 9:</b> Testing Label and Cosmetic GUI Controls	<b>319</b>
<b>Chapter 10:</b> Testing a TextBox Control with Input from a Keyboard	<b>333</b>
<b>Chapter 11:</b> Testing RadioButton and CheckBox Controls	<b>353</b>
<b>Chapter 12:</b> Menu Clicking for GUI Test Automation	<b>367</b>
<b>Chapter 13:</b> User-Defined and COM-Based Controls	<b>389</b>
<b>Chapter 14:</b> Testing Issues for Non .NET Applications	<b>411</b>
<i>Selected Bibliography</i>	<i>429</i>
<i>Index</i>	<i>431</i>

# Contents

---

*Introduction*

*xiv*

<b>Chapter 1</b>	<b>GUI Testing: An Overview</b>	<b>1</b>
	Unique Features of GUI Testing	4
	Developing an Automated GUI Testing Tool	5
	Expectation of Automated Testing	6
	Automated Test Teams	8
	How to Automate GUI Testing	10
	GUI Testing and Script Languages	12
	Using C# for Automation	14
	Test Scripts	16
	Summary	17
<b>Chapter 2</b>	<b>Available GUI Testing Tools vs. the Proposed Tool</b>	<b>19</b>
	Current GUI Testing Infrastructures	20
	Capture/Playback Is Not Automatic	20
	Test Monkeys	22
	Intelligent Automation	24
	Automatic GUI Testing Tools in the Marketplace	24
	CompuWare TestPartner	25
	IBM Rational Test Tools	25
	Mercury Interactive Tools	26
	Segue's SilkTest	27
	Open Source GUI Test Tools	27
	Advantages and Disadvantages of the Commercial Testing Tools	28
	Computer-Assisted GUI Testing	28
	The Common Features of Capture/Playback	29
	Editing the Recorded Test Script	30
	Implementing Testability Hooks	30
	Reusability for Regression Testing	31



	The Proposed GUI Testing Approach	31
	Active GUI Test Approach	31
	Generating Testing Data First	32
	Data-Driven Test Scripts	33
	Summary	34
<b>Chapter 3</b>	<b>C# Win32 API Programming and Test Monkeys</b>	<b>37</b>
	Understanding the Custom DLLs	38
	C# API Programming	40
	PInvoke Basics	40
	The <i>Marshal</i> Class and the <i>DllImport</i> Attribute	41
	Data Type Presentation	42
	A Simple C# API Example	43
	C# API Text Viewer	48
	A Base Class	49
	An API Utility Class	53
	<i>ConstantViewer</i> Class	57
	<i>DllImportViewer</i> Class	60
	<i>StructViewer</i> Class	68
	GUI of the C# API Text Viewer	75
	Starting with a Test Monkey	85
	A Dumb Test Monkey	85
	A Smarter Monkey	92
	Summary	97
<b>Chapter 4</b>	<b>Developing a GUI Test Library</b>	<b>99</b>
	GUI Test and the Custom <i>User32.dll</i>	100
	Exploring the <i>User32.dll</i>	102
	<i>Dumphbin.exe</i>	102
	Dependency Walker	104
	Building a Dynamic Linking Library for GUI Testing	105
	The Spy++ Tool	106
	Functions from the <i>User32.dll</i> for GUI Testing	107
	Expanding the Testing Capabilities	118
	A GUI Test Application	131
	Summary	135

---

<b>Chapter 5</b>	<b>.NET Programming and GUI Testing</b>	<b>137</b>
	XML Programming	138
	Writing XML Files	139
	Reading XML Files	142
	<i>XPathNavigator</i> Class	144
	<i>XmlDocument</i> Class	146
	Object Serialization	150
	Binary Serialization	150
	SOAP Serialization	156
	XML Serialization	159
	.NET <i>System.Collections</i> Namespace	161
	<i>Type</i> Class	162
	.NET <i>System.Reflection</i> Namespace	164
	Late Binding	167
	.NET <i>System.Threading</i> Namespace	170
	Summary	173
<b>Chapter 6</b>	<b>Testing a Windows Form in General</b>	<b>175</b>
	Overview of Software Architecture	176
	Presentation Layer	179
	Business Layer	179
	Data Layer	180
	GUI Components on the Presentation Layer	180
	Buttons	180
	ComboBoxes	181
	DialogBoxes	182
	Labels	182
	Menus	183
	TextBoxes	184
	Other Controls	184
	Expanding the GUI Test Library	185
	Methods for XML Accessibility and XML Serialization	187
	Methods for Late Binding	189
	Two Helper Classes	191
	Building a General Basis of a GUI Test Script	193
	Starting the Test Script Project	193

---

	Implementing the Command Buttons	194
	Coding the Timer Controls to Trigger GUI Events	197
	Visualizing the GUI Event Outcomes	201
	A Semi-Automatic Way of Verifying the Test Script	203
	Summary	207
<b>Chapter 7</b>	<b>Architecture and Implementation of the Automatic GUI Test Tool</b>	<b>209</b>
	Meeting Current and Future GUI Test Requirements	210
	The General Architecture of the Improved GUI Testing Tool	212
	Starting the AutomatedGUITest Project	213
	The Startup Form of the AutomatedGUITest Tool	213
	An Interface to Specify the Form of the Application under Test	215
	The Implementation of a <i>GUISurveyClass</i>	218
	Adding an Interface for Testing Data Confirmation	222
	Developing a General-Purpose GUI Test Script	228
	Putting Together the <i>AutomatedGUITest</i> Tool	240
	Starting the Application under Test	242
	Conducting a GUI Survey	245
	Specifying Data for the GUI Test	247
	Running the Test	252
	Conducting the First Automatic GUI Testing	261
	Summary	267
<b>Chapter 8</b>	<b>Methods of GUI Test Verification</b>	<b>269</b>
	Verification Requirements	270
	Front-end Actions and Back-end Business Functions	271
	Verifying Changes of an Application under Test	272
	Verifications Based on the Users' Interest	273
	Automated Verifications	274
	Simple Object Verification	275
	Specific Object Verification	276
	Lump Sum Verification	276
	Best Practice Verification	276
	Enhancing the AutomatedGUITest Tool	277
	Updating the <i>GUITestDataCollector</i> Interface	277
	Building a GUI Test Verification Class	287
	Updating the AutomatedGUITest Startup Form	301

---

	Updating the <i>GUITestScript</i> Class	305
	Adding the XML Document Viewer for Result Presentation	310
	Conducting a Fully Automated GUI Test	312
	Summary	318
<b>Chapter 9</b>	<b>Testing Label and Cosmetic GUI Controls</b>	<b>319</b>
	How to Test Label and Other Cosmetic Controls	320
	Upgrading the AutomatedGUITest Tool	321
	Adding a Method to the <i>GUITestVerification</i> Class	323
	Expanding the Testing Capability of the <i>GUITestScript</i> Class	323
	Updating the <i>GUITestActions</i> class	326
	Testing Cosmetic GUI Properties	328
	Collecting Testing Data	328
	Editing and Rerunning the Data Store	329
	Summary	331
<b>Chapter 10</b>	<b>Testing a TextBox Control with Input from a Keyboard</b>	<b>333</b>
	The <i>SendKeys</i> Class of the .NET Framework	334
	Code for Special Keys Using the <i>SendKeys</i> Class	334
	Methods of the <i>SendKeys</i> Class	336
	An Example to Connect to Your FTP Server	337
	Updating the Tool for Testing TextBox Controls	339
	Adding a Field to the <i>GUIInfo</i> Structure	340
	Modifying the Control Handling Methods in the <i>GUITestActions</i> Class	341
	Revising the Late Binding Method in the <i>AutomatedGUITest.GUITestScript</i> Class	342
	Adding a TextBox to the GUITestDataCollector Form	344
	Adding a Step to Send Keystrokes to a GUI Component	346
	Testing the C# API Text Viewer with the Updated Capabilities	350
	Summary	351
<b>Chapter 11</b>	<b>Testing RadioButton and CheckBox Controls</b>	<b>353</b>
	Characteristics of RadioButton and CheckBox Controls	354
	Updating the AutomatedGUITest Project	355
	Overloading a Method in the <i>GUITestUtility</i> Class	355

---

	Adding Code to the <i>TestExpectation</i> Class	357
	Enhancing the Testing Scope of the <i>GUITestScript</i> Class	358
	Testing RadionButton Controls	364
	Summary	365
<b>Chapter 12</b>	<b>Menu Clicking for GUI Test Automation</b>	<b>367</b>
	Characteristics of Menu Testing	368
	Window Handle and Menu Handle	369
	Menu and Submenu	369
	Class Name of Menu Windows	371
	Updating the <i>GUITestAction</i> Class with API Programming	371
	Marshaling a Few More Win32 Functions	372
	Adding Methods to Identify Menu Items	373
	Navigating a Menu Click Pathway	375
	Performing a Click on a Menu Item	377
	Updating the <i>GUITestActionLib.xml</i> Document	380
	Enabling Menu Survey Capability	381
	Enabling the AutomatedGUITest Tool for Menu Survey	382
	Invoking the Menu Survey Method	382
	Handling a Possible Error	383
	Updating the <i>GUITestScript</i> Class	384
	A Menu Testing Example	386
	Summary	388
<b>Chapter 13</b>	<b>User-Defined and COM-Based Controls</b>	<b>389</b>
	Basics of User-Defined GUI Controls	390
	Custom .NET GUI Controls	391
	ActiveX Component from Microsoft Visual Studio 6	393
	Needed Components for Testing User-Defined Controls	395
	Preparing a Testing Bench for Customized GUI Controls	395
	A New Method for the <i>GUITestUtility</i> Class	396
	Invoking the Test Bench from the AutomatedGUITest Tool	396
	Handling Exceptions in the <i>GUITestScript</i> Class	398
	Two More Examples	404
	Testing a Customized .NET GUI Control	404
	Testing an ActiveX GUI Control	406
	Summary	410

---

<b>Chapter 14</b>	<b>Testing Issues for Non .NET Applications</b>	<b>411</b>
	Adding a Method to Start Traditional Applications	413
	An Overview of the <i>System.Diagnostics</i> Namespace	413
	Updating the <i>GUITestUtility</i> Class	414
	Making the AutomatedGUITest Tool Probe the GUI Interface	416
	Adding a PictureBox Control as a GUI Probe	416
	Coding the Three Events for the GUI Probe	418
	Invoking a Non .NET Application for GUI Survey	421
	Updating the <i>GUITestScript</i> Class	422
	Putting the New Methods into Action	424
	Summary	426
	 <i>Selected Bibliography</i>	 429
	 <i>Index</i>	 431

# Introduction

---

**T**here are many books about software testing management. When they discuss software test automation, they introduce third-party testing tools. I have used many of the commercial software testing tools. Their developers declare that they have the capability to conduct various types of software tests and meet the requirements of an organization. But they have limitations. For example, many of GUI testing tools require users to record a series of mouse clicks and keystrokes. Others require users to write test scripts in a specified script language. Furthermore, the test scripts produced by these tools and methods need to be edited and debugged before they can be executed to perform the desired tests.

This book presents ideas for automating graphical user interface (GUI) testing. The sample code in this book forms a foundation for a fully automated GUI testing tool. Using this tool, users don't need to record, edit, and debug test scripts. Testers can spend their time creating testing cases and executing the testing.

## Who This Book Is For

Software engineers have long relied on the tools and infrastructures supplied by the current software testing tool vendors. Some engineers tell successful stories. But more engineers experience frustration. The tools are not automated enough, the tests are not efficient, the tests are not driven by data, and the test script generation and data composition methods need to be improved. One solution to software test automation is to develop testing tools instead of purchasing commercial tools based on the current inadequate infrastructure

This book is written for people who are involved in software engineering and want to automate the software testing process for their organizations. With the methods introduced in this book, software engineers should gain a good understanding of the limited automation provided by the available testing tools and how to improve the current test infrastructure and conduct a fully automated software test.

This book is for software engineers who want more effective ways to perform software tests. The automated testing tool introduced in this book can serve as an independent software test tool as well as an adjunct to the commercial tools.

To get the most out of this book, you should be a moderately experienced software developer and a test engineer in the process of conducting software tests for your organization. The explanations and examples in this book can be easily understood and followed by any

intermediate- to advanced-level programmer interested in expanding their knowledge in both software development and software testing.

This book's content includes programming techniques with examples in C#. Then it gradually progresses to the development of a fully automated GUI testing tool. Although the sample code is in C# using the Microsoft Windows platform, the tool has been evolved from a Visual Basic 6 project. I have also used these methods and developed a Java testing tool for a test project.

This book is also for the managers of organizations where software is developed and used. As economists have reported, software failures result in substantial economic loss to the United States each year. Approximately half of the losses occur within the software manufacturing industry. If you are a senior managerial administrator of a software organization, you are most likely interested in an improved software testing method. The other half of the losses is out of the pockets of the software end users. If your business or institution consists of software end users, you probably maintain teams to support the software purchased from the contract vendors. Being aware of testing methods will assist you with efficient software application in your organization.

## How This Book Is Organized

To present ideas well, the first two chapters in this book are an introduction to software testing, the available testing tools, and the expectations of software testers. Chapters 3 through 6 focus on Win32 API programming and .NET fundamentals needed for the development of the GUI test library. Chapters 7 through 14 are devoted to designing and implementing the Automated-GUITest tool until full automation is achieved. The following list includes a short description of each chapter.

Chapter 1, "GUI Testing: An Overview," describes the management of software GUI testing and the techniques built into .NET that can make it possible to test software dynamically.

Chapter 2, "Available GUI Testing Tools vs. the Proposed Tool," presents a brief review of some of the automated GUI testing tools on the market and the main points of the testing methods proposed in this book. The available tools have been proved by studies to be inadequate. The purpose of this chapter is to demonstrate the necessity of a new and improved test method, of adding more testing capabilities to a testing tool, and of creating a fully automated test for new and complex software projects.

Chapter 3, "C# Win32 API Programming and Test Monkeys," deals with how to marshal the Win32 API functions to be used as C# code for developing the tool. It also includes the code for you to develop a C# API Text Viewer to take the place of the old API Text Viewer for Visual Basic 6 programmers. The newly developed C# API Text Viewer will be used in three ways. First, it will generate C# code to marshal the Win32 functions needed for developing a



fully automated GUI testing tool. Second, it will serve as an application under test for the GUI testing tool as the testing tool is developed throughout the book. Third, the C# API Text Viewer will test the tool with different degrees of automation as the book progresses. In the end of this chapter, the C# API Text Viewer is used to present you with a test monkey.

Chapter 4, “Developing a GUI Test Library,” guides you through the development of a GUI test library with functions for finding GUI objects from an interface and manipulating the mouse actions.

Chapter 5, “.NET Programming and GUI Testing,” introduces .NET technologies—including reflection, late binding, serialization, and XML programming—with regard to data store, data presentation, and GUI testing.

Chapter 6, “Testing a Windows Form in General,” describes how a GUI test script is created manually. This handcrafted test script forms the base on which a fully automated GUI test will be built.

Chapter 7, “Architecture and Implementation of the Automatic GUI Test Tool,” lays out the architecture for a automated GUI testing tool and helps you build the first evolution of the proposed AutomatedGUITest tool.

Chapter 8, “Methods of GUI Test Verification,” explains how to effectively verify the test results and add code to the AutomatedGUITest project for automatic verification.

Chapter 9, “Testing Label and Cosmetic GUI Controls,” describes the processes of testing Label and other cosmetic GUI controls.

Chapter 10, “Testing a TextBox Control with Input from a Keyboard,” discusses the `SendKeys` class of .NET Framework and updates the AutomatedGUITest for testing TextBox controls.

Chapter 11, “Testing RadioButton and CheckBox Controls,” shows you how the RadioButton and CheckBox controls are used in software applications and discusses ways to test them.

Chapter 12, “Menu Clicking for GUI Test Automation,” introduces more Win32 API functions for discovering menu items and testing them.

Chapter 13, “User-Defined and COM-Based Controls,” introduces methods for developing custom GUI controls in the Microsoft Visual Studio .NET IDE and Microsoft Visual Studio 6 IDE and updates the AutomatedGUITest tool for testing such controls.

Chapter 14, “Testing Issues for Non .NET Applications,” presents the methods for testing an unmanaged application. This chapter tests `Notepad.exe` as an example after new code is added to the tool project.

## About the Examples

The examples start with the programming precepts of C# and Win32 API functions. The goal is to use the predefined methods of a programming language to complete an AutomatedGUITest tool project and avoid reinventing the wheel. There are four kinds of sample code in the chapters:

- Sample code for developing a C# API Text Viewer
- Simple examples to demonstrate using C# and Win32 API functions
- Example projects to be tested by the AutomatedGUITest tool
- Sample code of the AutomatedGUITest tool project

The code examples in the first category most often appear in Chapters 3 through Chapter 6. Thereafter, Chapters 7 through 14 are totally dedicated to automating the test project. The sample code in these chapters is in the third category. There are only three examples of the second category, simulating real assemblies under test. They are implemented in Chapters 3 and 12 and submitted to testing throughout the book.

Besides the C# API Text Viewer, the code in Chapter 3 also develops a test monkey. Chapter 5 develops an XML document viewer that will be used to present the test results for the AutomatedGUITest tool.

In Chapter 4 and thereafter, some code is added to the GUI test library and AutomatedGUITest project in each chapter. At the end of each chapter, the sample code can be compiled to produce an executable assembly. The testing tool achieves different degrees of automation until a fully automated test tool is developed by the end of the book.

## Where to Find the Source Code

The sample and project code for each chapter can be downloaded from [www.sybex.com](http://www.sybex.com) by performing a search using this book's title, the author's name, or the ISBN (4351). This saves you from having to type in the code. It also includes a complete compiled version of the project, which allows you to put the AutomatedGUITest into immediate practice for your software project.

To execute the AutomatedGUITest.exe file, you can copy the files from the Chapter14\AutomatedGUITest\bin\Debug folder of the downloaded sample code to a computer system. The minimum requirements for a computer system are as follows:

- Windows 95/98/2000/NT/XP
- Preinstalled .NET Framework
- 20MB of free hard disk space

If you are still using earlier versions of the Microsoft Visual Studio .NET integrated development environment (IDE) at this point (older than Microsoft Visual Studio .NET 2003 IDE), you will not be able to open the sample projects directly. The problem is that the C# project files with the extension `.csproj` are incompatible with earlier versions of the Microsoft Visual Studio .NET IDE. To use the code, you can follow the procedures in each chapter to create new projects and include the code files with extensions of `.cs` downloaded into your projects.

Although the sample code in this book is developed under the Microsoft Visual Studio .NET 2003 IDE, there are other open-source .NET IDEs available through free download:

**Eclipsing .NET** IBM released the Eclipse .NET to the open-source community. This product works on Windows 95/98/NT/2000/XP. You can download the components for the Eclipse .NET from [www.eclipse.org](http://www.eclipse.org). After downloading the `eclipse-SDK-2.1.1-win32.zip` file, install it with the combination of the Microsoft .NET SDK, which is also a free download from [msdn.microsoft.com/netframework/technologyinfo/howtoget/default.aspx](http://msdn.microsoft.com/netframework/technologyinfo/howtoget/default.aspx). Then get the open-source C# plug-in through the Eclipse .NET IDE. An article at [www.sys-con.com/webservices/articleprint.cfm?id=360](http://www.sys-con.com/webservices/articleprint.cfm?id=360) introduces the downloading and installation in detail.

**#develop** Short for SharpDevelop, this is another open-source IDE for C# and VB.NET on Microsoft's .NET platform. You can download #develop from [www.icsharpcode.net/OpenSource/SD/Default.aspx](http://www.icsharpcode.net/OpenSource/SD/Default.aspx).

**DotGNU Portable .NET** This open-source tool includes a C# compiler, an assembler, and a runtime engineer. The initial platform was GNU/Linux. It also works on Windows, Solaris, NetBSD, FreeBSD, and MacOS X. You can download this product from [www.southern-storm.com.au/portable\\_net.html](http://www.southern-storm.com.au/portable_net.html).

## CHAPTER I

---

# GUI Testing: An Overview



The saturation of software in industry, educational institutions, and other enterprises and organizations is a fact of modern life almost too obvious to mention. Nearly all of the businesses in the United States and in most parts of the world depend upon the software industry for product development, production, marketing, support, and services. Reducing the cost of software development and improving software quality are important for the software industry. Organizations are looking for better ways to conduct software testing before they release their products.

Innovations in the field of software testing have improved the techniques of writing test scripts, generating test cases, and introducing test automation for unit testing, white box testing, black box testing, and other types of testing. Software testing has evolved over the years to incorporate tools for test automation. Mercury Interactive, Rational Software of IBM, and Segue are a few of the main tool vendors. The purpose of these tools is to speed up the software development life cycle, to find as many bugs as possible before releasing the products, to reduce the cost of software development by automating effective regression testing, and to increase application reliability. The apparent goal of the software testing tools is to generate test scripts that simulate users operating the application under development. Usually test engineers are trained by the tool manufacturers to write test scripts by hand or to use a capture/playback procedure to record test scripts. Both writing and recording test scripts is labor intensive and error prone. The potential of the available tools to reduce the manual, repetitive, labor-intensive effort required to test software is severely limited.

Recently, organizations have purchased the commercial testing tools but the test engineers can't use them because of their inadequate test infrastructure. Often these tools don't have the capabilities to handle the complexity of advanced software projects, aren't capable of keeping up with technology advancements, and aren't diverse enough to recognize the varieties of third-party components in today's software development business. Needless to say, profitable organizations have trade secrets the commercial testing tools are not aware of. Because of these inadequacies, the United States loses \$59.5 billion each year due to the bugs in the software not detected by the current testing means (Tassej 2003). Test engineers and experts are working to develop more effective testing tools for their organizations. Thus, they can improve the current test infrastructure with regard to the following:

- Enhanced integration and interoperability testing
- Increased efficiency of testing case generation
- Improved test code generation with full automation
- A rigorous method for determining when a product is good enough to release
- Available performance metrics and testing measuring procedures

It is estimated that an improved testing infrastructure should try to reduce the current software bugs by one-third, which can save the United States \$22.2 billion. Each year, in order

---

to achieve such a software testing infrastructure, researchers have published their findings. For example, in my previous book, *Effective Software Test Automation: Developing an Automated Software Testing Tool* (Sybex 2004), I discussed the methods to develop a testing tool that can be adapted to various software programming languages and platforms. The outcome of the book was a fully automated testing tool. It automated the entire process of software testing—from data generation and test script generation to bug presentation—and broke a number of bottlenecks of the current testing tools in the following ways:

- *By actively looking for components to test instead of recording test scenarios by capture/playback.* This approach can enable the test script to conduct a thorough testing of an application. It is reported that testing cases are more effective in finding bugs after the test script is developed. Thus, testers don't spend time writing test scripts; instead, they spend their time studying and creating multiple testing cases.
- *By effectively generating one script to test all the members (constructors, properties, and methods) of a DLL or an EXE, including members inherited from base classes.* This test script prepares a real and complete object that can be reused for integration testing and regression testing. Therefore, the number of test scripts for one software project can be reduced, making the management of test scripts easy.
- *By effectively reusing the previously built test scripts of lower-level modules for testing higher-level modules with a bottom-up approach.* This helps you avoid stubbing and using mock objects to conduct integration testing.
- *By automatically writing test scripts in a language used by the developers.* The testers don't have to learn a new script language. The developers can understand the test scripts. This enhances the collaboration between testers and developers.
- *By innovatively integrating automated script generation with a Unified Modeling Language (UML).* The project can generate test scripts early when a detailed design is available. The test script can be executed for regression testing and reused for integration testing throughout the development life cycle. Because the test can be conducted nightly in an unattended mode, it guarantees that all the newly developed code will be tested at least once at any stage.
- *By automatically compiling and executing the generated test scripts without editing or debugging.* If the design of the software architecture is modified in middle of the development process, a new test script can be automatically generated immediately with accuracy.

However, due to the uniqueness of the graphical user interface (GUI) components in software products, techniques and programming methods for developing an effective automated GUI testing tool required a separate book. We will use the same philosophy used for automating the non-GUI testing in the preceding book and develop a fully automated GUI testing tool; that is, the users feed the GUI testing tool with an application and get the defect (bug) reports.

## Unique Features of GUI Testing

Early software applications were command-line driven. Users remembered and typed in a command and the system performed some actions. A more effective application could display possible commands on the screen and then prompt the user for the next command. Nowadays, the software industry is in the windowing system era. Virtually all applications are operated through graphical user interfaces (GUIs). GUI components are often referred to as windows, icons, menus, and pointers. Users drag a mouse, click buttons, and apply various combinations of keystrokes. The applications are triggered to perform desired actions by the mouse events and keystroke events. Thus, GUIs have made the software friendlier to users.

Software verification and validation through testing is a major building block of the quality assurance process. GUI tests are vital because they are performed from the view of the end users of the application. Because the functionality of the application is invoked through the graphical user interface, GUI tests can cover the entire application,

To automatically test software before the GUI era, testers relied on test scripts with a collection of line commands. The executions of the programs were independent from the screen status. Testing GUI components is different and more difficult because it requires a script to reassign the input stream, to click buttons, to move the pointer, and to press keys. Then the scripts need to have mechanisms to record the responses and changes of the dynamic states of the software. Comparing the responses and changes with the expected baselines, the scripts are able to report bugs.

There are some specialist tools available to test GUI-based applications. They often come with a variety of features. Testers expect that these features will enable them to conduct GUI testing with regard to platform differentiation, recognition of GUI components, test script automation, synchronization, result verification, and easy management. The currently available tools are very much influenced by platforms. For example, the 32-bit Microsoft Windows operating system (Win32) is currently the dominant platform for development. Testing tools developed for Win32 can not be used with other platforms, such as the Unix, Linux, and Macintosh operating systems.

The broadly accepted method today of generating GUI test scripts relies on the capture/playback technique. With this technique, testers are required to perform labor-intensive interaction with the GUI via mouse events and keystrokes. The script records these events and later plays them back in the form of an automated test. Test scripts produced by this method are often hard-coded. When different inputs are needed to conduct the test, the test script needs to be re-created. Regression testing using these test scripts is also impractical when the GUI components are under development. It is often hard to generate all the possible test cases for all of the GUI components and conduct a thorough test. Human interaction with

the GUI often results in mistakes. Thus, the capture/playback procedure records the redundant and wrong clicks and keystrokes.

Based on the current testing technologies, GUI test automation invariably requires manual script writing, editing, and debugging. On one hand, tool vendors keep on informing testers of the powerful functions their tools are capable of. On the other hand, testers must face a great number of technical challenges getting a tool to work with the application under test (AUT). It is well known that GUI components are modified and redefined throughout the development process. The generated test scripts are unable to keep up with design changes.

Compared with non-GUI software testing, a GUI test infrastructure based on the current testing tools is also expensive with regard to purchasing the tools and the efforts that must be made to maintain the tests. Very often, only a part of the GUI components are tested automatically. The other part still requires manual tests. In the following sections, I'll address the inadequacy of GUI testing by discussing an approach for improving the current test infrastructure.

## **Developing an Automated GUI Testing Tool**

The raw GUI test scripts recorded by the capture/playback technique perform the apparent mouse and key actions if they don't fail to execute. But even when the execution succeeds, the captured results don't verify whether the invocation of the business functions (often, non-GUI modules) caused by the GUI events is correct. Complicated graphic output must be tested. The GUI test scripts don't test the input variation to the GUI components. Without manual editing and debugging, these test scripts are not able to test whether the GUI components are created and drawn as desired.

The purpose of this book is to show how to use more effective programming methods to develop a GUI testing tool. This tool will help you avoid the drawbacks and pitfalls of the current testing methods. The generated test script should be able to capture the event actions and changes performed on the GUI component being tested and the invocation of the related non-GUI components caused by the GUI events. It will also verify the position and the appearance of the GUI components and test the input to the GUI components. The result of the event actions and invocations will be converted from GUI to readable text formats. Then the test result reports will be used by the developers to fix the detected bugs.

To achieve an improved test infrastructure, this book will discuss ways to develop GUI testing tools that can generate flawless test scripts. These scripts will have functions to test various aspects of the GUI components. They will not become obsolete due to changes of the GUI components. If more GUI controls are added or reduced, a new test can be regenerated by feeding the modified application to the tool.



In this book, I'll also discuss methods for developing testing tools that require minimum training. The current testing tools have complicated user interfaces and users must be trained to use them. Almost every GUI testing tool has a script language for writing test scripts. Developers are often isolated from software testing automation by the language differences. In this book you'll learn how to develop a tool to write test scripts in the same language that is used by the developers in an organization. Thus, the testing tool, the application under test, and the automatically generated test scripts are all developed in one language. The test projects become more readable and maintainable and bring testers and developers together.

I'll also discuss methods for generating testing cases so the combination of the testing cases and scripts will be more effective in finding bugs. None of the tools on the market that rely on the capture/playback or reverse-engineering technique is able to locate a GUI component and write a relevant test script for it automatically. Thus, we need a method to conduct an automatic and exhaustive survey of the GUI components and write test scripts and test data with respect to different testing issues of the components.

Finally, using these methods to develop a testing tool will save time and money, deliver accurate products, and make the job interesting for the organization.

## **Expectation of Automated Testing**

Automated tests help to greatly reduce the time and the cost spent on software testing throughout the entire development life cycle. Furthermore, automation ensures that tests are performed regularly and consistently, resulting in early error detection that leads to enhanced quality and shorter time to market.

Current testing techniques are not able to automate the GUI testing process before an executable test script is created. The tester is often required to use a capture/playback method to record scripts. The recording process is in fact a labor-intensive interaction between the tester and the GUI. If there is a bug the process fails to record and one bug is detected during testing, the tester must repeat and continue the script recording process until the bug is fixed. Thus, bugs are detected one by one throughout manual script recording. When the recorded script is ready to be executed, the possibility of finding bugs by replaying it is limited.

Testers expect the testing tools to actively find GUI components, generate the respective testing data, and use the generated data to drive the test script generation and execution. There is not a tool that is capable of conducting an active survey for the existing GUI components of an application. Although vendors often claim their tool is capable of data-driven script generation, testers often need to start a wizard and enter testing data. This wizard-driven data generation is also labor intensive and error prone. The first execution of this data is not effective in detecting bugs. To find bugs, one test script should be able to test against many test data sheets. Testers expect a fully automated method to generate multiple copies of testing data.

Commercially available testing tools are very much platform dependent. They often have their own testing environment. A recorded test script usually runs in the environment in which it has been recorded. Tools also are accompanied by tool-dependent script languages. Thus, software testing is often limited by the tester's system. If developers and other personnel need to be involved in testing, more tool licenses need to be purchased from the tool vendors. This limits the test script portability.

Researchers and tool developers are making an effort to achieve a fully automated testing tool that will recognize the GUI components and generate testing data and test scripts in sequence. It should also be capable of being executed for regression testing unattended. Based on the facts reported in *The Economic Impacts of Inadequate Infrastructure for Software Testing* (Tassey 2003), the current approaches to testing need to be revised or improved, and some even need to be discarded. We will therefore heed the experts' advice to develop a testing tool. A new tool capable of locating bugs more effectively will have the following features:

- *The test script language will be the same one used to develop the application under test.* Users won't be required to manually click mouse buttons, press keys, and record test scripts. Thus, testing can be run on the tester's systems as well as on the developer's systems. The capability of executing the test on user's systems will be helpful for the alpha and beta version release of the application.
- *Management of software testing will become easy.* Because the tool will actively search for GUI components and write test scripts, a maximized test will be achieved and the redundancy of test script generation will be reduced. One test script will be capable of testing against an optimal number of testing cases.
- *The developed tool will have an open architecture that makes it flexible and easily modified for extended testing functionality.* Upgrading of the tool can be accomplished within the resource limits of an organization. Thus the tool will always be compatible with the advancement of technologies and the complexity of the software projects. It will also be able to recognize third-party GUI components, custom controls, and generic objects for test data and script generation.
- *Regression testing will be fully automated.* Test execution has always been a boring, mechanical, repetitive, dreary aspect of testing. With this tool, the regression testing process can be conducted nightly and daily without human attention.
- *Test results will be reported and stored in a widely accepted format (e.g., a popular spreadsheet program, an XML document, or an HTML format).* When a bug is found, the report will pinpoint the problem in the source code. Users don't need to be trained in order to understand and use the report to fix defects. In addition, an Internet bug tracing system can be enabled.

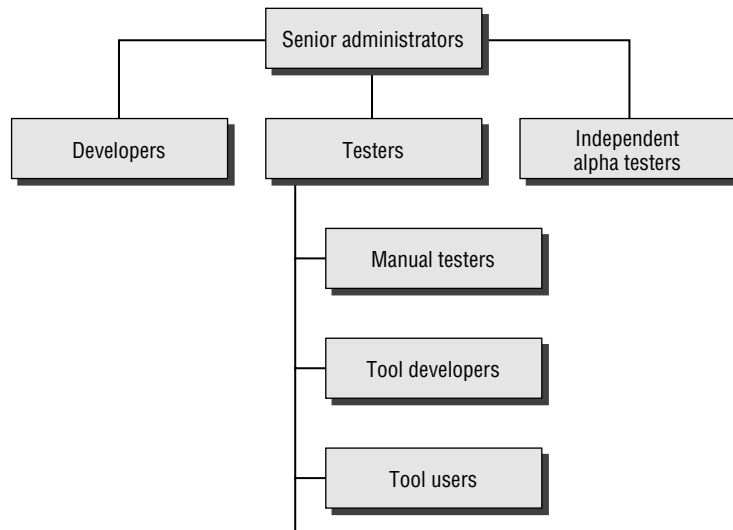
## Automated Test Teams

We all agree that with a properly automated testing process, more testing cases can be conducted and more bugs can be found within a short period of time. Charles Schwab & Co., a major U.S. stockbroker, reported that, based on the current testing means, a typical system test required 52 hours of effort manually but only 3 hours after automation. Automated Data Processing (ADP) reports that its elapsed time for testing has been reduced 60% through automation. Studies indicate that the number of defects discovered by tool users increase by 10% to 50%. Many organizations today rely on automated testing tools for the high quality of their products.

On the other hand, a survey in 250 organizations found that only 35% of testers were still using automated tools one year after the tool installation due to the reported testing inadequacy. To avoid turning back to the tedious and time-consuming manual testing method, many organizations train their staff to develop more effective automated tests. Figure 1.1 shows an organizational chart for a test team. An effective test team must involve senior administrators interested in quality products and a high degree of test automation. Developers should be willing to share and convey knowledge with regard to testing issues. Tool developers develop more effective testing tools based on the experience of manual testers and tool users. Alpha testers work independently but share test scripts (automated or manual) with other testers.

**FIGURE 1.1**

The organizational chart of a test team with the components of senior administrators, supportive developers, testers, and alpha testers



An automated testing process is the result of the evolution and maturity of manual testing efforts. During the tool development life cycle, manual testers continuously make contributions of automation strategies and testing techniques. Thus, before the test of a new component can be automated, manual testing should be conducted and verified. Then the optimal testing method can be implemented and integrated into the automated tool. As the testing methods are accumulated, the tool will become useful for future projects instead of just the current project.

Members of a conventional automated test team are tool users. A new automated test team would ideally include tool developers, tool users, and manual testers. Manual testers analyze project risks, identify project requirements, develop the documentation to design the overall testing scenarios, and specify the test data requirements to achieve the needed test coverage. The manual test engineers also prioritize testing jobs to be integrated into the automated testing tool.

The management strategy for developing a testing tool should resemble the management strategy used in the development of other applications. First, the team will select a development model and environment. The project described in this book will be implemented in the same language as the application under test. This will enhance the collaboration between the testers and the developers. Thus, testers will have the knowledge they need to test the application and make effective testing cases, and developers will make suggestions for improving the tool development and test script generation. The developers can use the tool to complete the unit and integration testing at code time.

Next, requirements need to be collected based on experiences that other testers, developers, and all kinds of end users have had with manual testing. The tool project specification is a cooperative effort between managers of all levels, product learning personnel, testers, and developers. An object-oriented design document is also important for the success of the tool's reusable architecture. This document will not only be used to complete the current implementation, future upgrading will also be based on it. The tool developers should have broad knowledge of software engineering and software testing and automation and programming skills. When it is implemented, the code will need to be tested. Bugs should be found and fixed so that they will not be introduced into the applications under test. Finally, the automated testing team needs to create training materials for the end users. The team can recruit the developers, engineers, and other personnel to test the application. Within a short period of time, maximum testing will be conducted. As many bugs as possible will be detected and fixed, thus assuring high-quality code.

The automated test team benefits the current project as well as future projects because the tool will be maintained. Unlike commercial testing tools, this tool will always keep up with current technology advancements, adapt to project complexity, and be able to test the creative and innovative aspects of an application.

## How to Automate GUI Testing

The degree of automation for the tool depends on the testing requirements collected for an organization. With the knowledge of what GUI components to search for, the automated test team can implement drivers to create test scripts in various situations that correspond to the testing scenarios of the manual testing. A team with skillful manual testers can develop a tool with a higher degree of automation. In that case, the tool development will require more investment, but because the technologies of the software industry change quickly, developing a highly automated tool quickly pays off by increasing productivity and quality.

The starting point of testing an application's GUI is to locate the GUI components. Currently available tools depend on mouse events and keyboard actions from users. A fully automated testing tool should recognize these GUIs programmatically. We need to enable the tool to move the pointer automatically to conduct a GUI component survey. If a GUI component is detected at the movement of the pointer, the tool will identify it by its unique properties, check its status, generate testing data, and write code to activate the GUI and catch the consequences. These consequences could be visible (external GUI level) or invisible (internal non-GUI modules) and are stored in the test results.

A GUI component can be recognized by its name, its position in hierarchy, its component class, the title of its parent window, and the programmer-assigned tag or ID. During each session of running an application, the GUI components are located with unique pairs of coordinates on the screen. However, GUI positions are often under modification by programmers, platforms, and screen resolution. A GUI test script should avoid hard-coding the position for tracking the GUI components. The component name is also not useful for writing test scripts. When we write a test script for testing a GUI component, we often find that programmers assign one name to several GUI components. In other cases, the component names are assigned automatically by the integrated development environment (IDE).

Usually, the programmer-assigned properties (label text, button captions, window titles, etc.) are unique and can be effectively used to identify a GUI component. We can develop our automated test tool to use the combination of the title of the parent window, the name of the component class, the caption, and other custom properties to locate the GUI components using test scripts.

After the survey and the identification of the GUI components, test scripts should be created. Current testing tools don't have the capability to generate test scripts automatically. When a tool has a capture/playback or a reverse-engineering processor to generate test scripts, the vendor claims it is a powerful testing tool. Testers in some teams don't trust the test scripts recorded by the capture/playback or reverse-engineering procedure, so they write test scripts from scratch. Because the test script can invoke the application, vendors claim this testing method is automatic. Thus, the testers are often found spending most of their time writing test

scripts or playing with the capture/playback tool to edit and debug the generated test scripts. They don't have enough time to manually create testing data, which is time consuming and requires creativity.

However, more and more testers disagree that the capture/playback tools are automated testing tools. We need to develop our own testing tools with full automation. My book *Effective Software Test Automation: Developing an Automated Software Testing Tool* discussed technologies such as reflection, late binding, CodeDom and testing data generation. The result of the book was the creation of a fully automated testing tool for non-GUI components. This book will use an active GUI survey approach to collect GUI test data, and the collected data will drive the test. In order to conduct an automatic GUI survey, this book will also discuss some Win32 applications programming interface (API) techniques. During the GUI survey, some API-implemented functions will simulate a person manipulating the mouse and the keyboard. Reflection techniques can be used to dismantle the component under examination. Late binding can invoke related members of the GUI event. The generation of test data and scripts becomes fast. Furthermore, the survey and script generation will be done automatically without the human tester's attention. Testers will spend most of their time composing effective testing cases. The automatically generated test script will test the component against multiple copies of testing cases and thus maximize the bug finding process.

Based on the preceding discussion, current automated test teams have a lot of manual testing responsibilities because the commercially available automated testing tools are inadequate in data and script generation and the recorded raw scripts lack the capability to verify test results. In addition to a full test automation, the developed tool will have the following advantages:

**Less intrusive** An intrusive tool requires developers to implement testability hooks in the application. These hooks are not desired by end users. Using an intrusive tool and using extra coding in the application could cause problems in the future. GUI testing methods without capture/playback require less human interaction with the application, making the tool less intrusive.

**Platform and language independent** Although the sample code in this book will be written in one language within the Windows operating system, the method can be extended to any other environments and coded in different languages. There is also no need to purchase additional hardware to handle the testing overhead.

**Flexibility for upgrading** Software testing is influenced by continuous evolution and addition of new technology, including advances in computer science, new testing technologies, and new development methods from outside sources. In addition, organizations continuously strive to create profitable new techniques and products. When a technology is implemented in an organization, the testing tool will be able to test the new features. If the tool lacks the capability to test new features, developers can upgrade the tool.