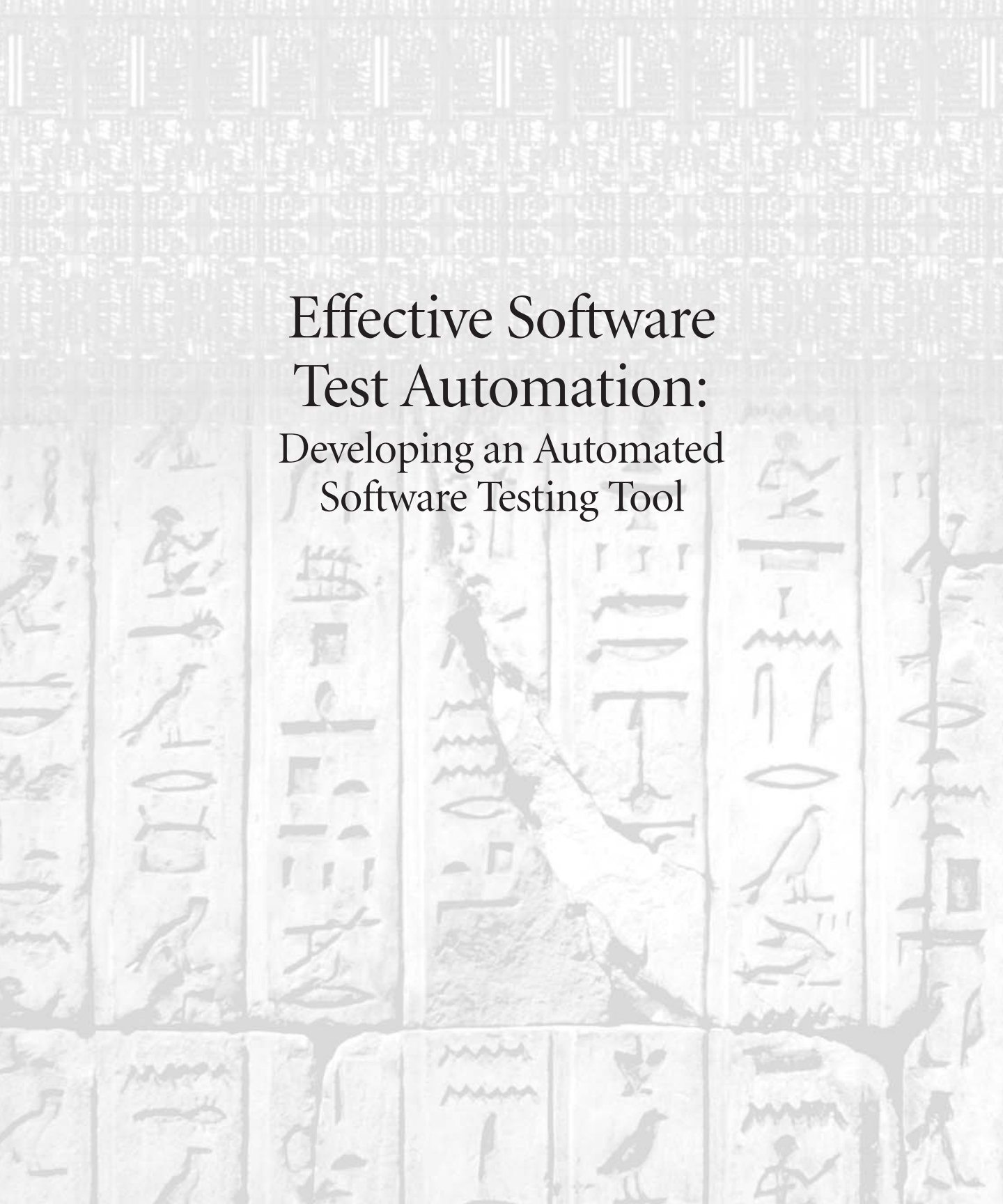

Effective Software Test Automation:

Developing an Automated Software Testing Tool

Effective Software Test Automation:

Developing an Automated Software Testing Tool

Kanglin Li and Mengqi Wu

San Francisco · London

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

Library of Congress Card Number: 2003115585

ISBN: 0-7821-4320-2

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. 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.

I dedicate this book to my father and my mother, Li Zuoxi and OuYang Meifang, who have worked hard and are still working to bring us up. They now live in Leiyang, China, a beautiful and small city where the first people who knew how to make paper in the world lived 1900 years ago. To my sisters, Li Liangyu and Li Xingyu; my brother, Li Chungui; and their spouses, Deng Yousheng, Luo Meilong, and Liu Wang. To my nephews, Deng Hao, Luo Yuanzhao; and Li Chuteng. Finally, to my wife, Mengqi Wu, who made a contribution to this book. She is a system engineer at Lucent Technologies and is also the coauthor.

I also dedicate this work to the coworkers who lost their jobs during the economic hard times.

—Kanglin Li
Des Moines, IA

Acknowledgements

I hope you find this book a valuable reference for software development, especially for software test automation, of which the goal is to reduce defects and costs and enhance the quality and profitability of software products.

I extend my first thank to Tom Cirtin, the acquisitions and developmental editor who made this book available to you. He has worked diligently with wisdom, enthusiasm, and inspiration. He is the best. I also thank Jordan Gold.

Other members of the Sybex team include Acey J. Bunch, Judy Flynn, Erica Yee, and others. They have done a great job of technical editing, copyediting, and production editing with suggestions and comments. I want to thank them all for their good and hard work.

I also express my appreciation to my former coworkers at Agilent Technologies, Inc. for the good times we shared and the good work we did: Susan Powell, Kenneth Ward, Kevin Keirn, Chuck Heller, Pat Kennedy, Greg Kuziej, Phil Driggers, Jun Liu, Ting Wu, Jing-Dong Zhang, Dave Willis, Alicia, Jerry Metz, Mike Hawes, and Jayson Jackman. At this point, I also want to thank Marcie and Bob Hervey and Steve and Molly Child and their families for their friendship.

Special gratitude is extended to Mr. Ray Gonzalas, a software engineer from National Security Agency. He was one of the first people who read the original manuscript, word for word, for this book. He made valuable comments, numerous corrections, and constructive suggestions. The current rendition of this book reflects his hard work. Special thanks also goes to Lena Berrios. Another among the manuscript's first reviewers, she did a lot of editing to polish the language and improve the writing style. I worried that, as a professional librarian and linguist, she might be a bit bored reading a computer book. Fortunately, she told me she learned from it and encouraged me to finish this project.

Sincere thanks are extended to my graduate advisors, Dr. Aziz Amoozegar at North Carolina State University and Dr. M.R. Reddy at North Carolina A&T State University, who taught me how to think and how to write.

Finally I'd like to express my indebtedness to Dr. Gregory Tassey from the National Institute of Standards and Technology, who gave me the permission to use the facts and data of his published study.

Contents at a Glance

Contents

Introduction

There are many books about software testing management. When they discuss software test automation, they introduce third-party testing tools. This book describes techniques for developing a fully automated software testing tool. You can use this tool to generate test scripts for continuous unit testing, integration testing, and regression testing.

Software defects are common and cause economic losses from time to time. Today, software organizations invest more time and resources in analyzing and testing software as a unit rather than as independent entities. Software engineers have observed that writing testing code is as expensive and time consuming as developing the product itself. To ensure software quality, organizations encourage software developers and testers to achieve objectives such as these:

- Locating the source of defects faster and more precisely
- Detecting bugs earlier in the software development life cycle
- Removing more defects before the product is released

Improved testing tools can reduce the cost of software development and increase the quality of software. An automated testing tool must have the following characteristics:

- Accurate functionality, reliability, interoperability, and compliance
- An interface that is user friendly and easy to learn and operate
- Enhanced fault tolerance and automatic error recoverability
- Efficient algorithm for time and resource management
- Stable and mature final products that can be maintained and upgraded
- Easy portability with regard to installation, uninstallation, adaptability, and security

I have used many of the commercial software test 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, some of them require users to record a series of mouse clicks and keystrokes. Others require users to write test scripts in a specified script language or to generate a test script automatically to test only one function (member) of a software module. 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. Automatic generation of the testing data is beyond the reach of these tools, and integration testing involves extensive manual stubbing and guesswork.

Software test engineers would like to see a fully automated software test tool on the market, one that is capable of completing testing tasks from generating test scripts and composing the

testing cases to presenting the results and fixing the bugs. But the tool vendors are not able to keep up with the complexity and technology advancements in today's software projects. In addition, software products can include features that incorporate a company's trade secrets, which the commercial testing tools won't have the capability of testing. Engineers are often in the position of having to develop their own tools to cover the gaps. This book presents a way to develop and enhance a testing tool development with full automation.

When I was trained to use commercial tools, the trainers from the manufacturers presented hundreds of testing features. Software test engineers do appreciate these features, and they are important in improving the quality of software. But the tedious and time-consuming processes of editing and debugging the generated test scripts sometimes prevent a thorough software test. Thus, software products are delivered to end users with costly errors. These costs are shared by virtually all businesses in the United States that depend on software for their development, production, distribution, and after-sales supports and services. To address these current inadequacies, this book will introduce an automated method to minimize the data editing steps, generate a test script to test the entire application, and free you from having to edit and debug the test script manually. The final product simply accepts an application under test and delivers the test results.

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 frustrations. The automation is not enough, the test is not efficient, and the test script generation and data composition methods need to be improved. One expert's solution to software test automation is to develop testing tools instead of purchasing commercial tools developed with 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 by 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 test tool introduced in this book can serve as an independent software test tool as well as an adjunct to the commercial tools.

I assume you are a moderately experienced software developer and a test engineer in the process of conducting software test 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.

Knowledge of the fundamentals of software testing is essential for software test engineers. Examining a combination of programming and testing issues leads to a solid solution to software test automation. This book's content includes sound programming techniques with examples in C#. Then it gradually progresses to the development of a fully automated test tool. Although the sample code is in C# using the Microsoft Windows platform, the concept can be used with other languages and platforms.

As economists have reported, software failures result in a 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 test method. The other half of the loss comes 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

In order to present ideas well, the first two chapters introduce the techniques of software testing and programming languages. Chapter 2 also lists advantages and disadvantages of the current testing tools and defines the objectives to overcoming the disadvantages. Chapters 3 through 6 focus on .NET programming related to the development of the AutomatedTest tool. During the fundamental discussions, as the knowledge becomes available and the time is appropriate, progress will be made on the AutomatedTest tool project. After the requisite background is covered, Chapters 7 through 11 continue to the completion of the AutomatedTest tool, using the programming techniques from chapters 3 through 6. At this point, the AutomatedTest tool will be able to generate test script to conduct unit testing, integration testing, and regression testing It will also report defects found. At last, Chapter 12 concludes this book with examples of testing some real life software.

The book is organized as follows:

Chapter 1, "Software Testing: An Overview," describes the techniques built into .NET that can make it possible to test software dynamically and illustrates how to integrate these techniques into an automated software test process.

Chapter 2, "Current Testing Infrastructure vs. the Proposed Testing Methods," presents a brief review of some of the automated 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 power of the new test method, of adding more testing capabilities to a testing tool, and of creating a fully automated test for new

and complex software projects. The available tools test software in various environments (for example, Java, Unix, Linux, and Windows). Thus, the methods introduced are not limited to Microsoft platforms and products. They can be extended to other development environments.

Chapter 3, ".NET Namespaces and Classes for Software Testing," deals with the starting points of an automated testing task.

Chapter 4, ".NET Reflection for Test Automation," introduces the .NET Reflection namespace. The Reflection namespace can reflect the software under test as a prism reflects the sunlight.

Chapter 5, "Spreadsheets and XML for Test Data Stores," introduces a method to store testing cases inside an XML document at coding time. It also discusses how to program an MS Excel worksheet in the test script to store the test information and present test results.

Chapter 6, ".NET CodeDom Namespace," describes the method for using the .NET CodeDom namespace to generate test scripts based on the information revealed by the Reflection namespace.

Chapter 7, "Generating Test Scripts," implements the method using the .NET CodeDom to write test scripts based on classes, methods, and parameters dynamically. This method is able to generate one test script to test all members of an assembly. The advantage of generating one test script for an assembly is that the test script can return a complete and full object of the class under test after the execution. The returned object can be reused for integration testing or for testing a method passing parameters by objects.

Chapter 8, "Integration Testing," explores a method that will automatically test parameters passed by objects and conduct integration testing. Instead of using the available stubbing or mock objects methods, this book introduces a bottom-up approach to reuse the previously generated test script to automate integration testing.

Chapter 9, "Verification, Validation, and Presentation," describes the processes of the automatic test verification, validation, and result presentation. It also introduces a method showing how to use this tool to test software that is still in development, at its earliest available design. The result of the verification, validation, and bugs found are presented with a spreadsheet.

Chapter 10, "Finalizing the AutomatedTest Tool," wraps up the test tool development and shows how to improve the appearance of the graphical user interface.

Chapter 11, "Updating the AutomatedTest Tool for Testing the Windows Registry," shows you how to update the AutomatedTest tool for testing against the Windows system Registry. Thus, the readers can add more testing capabilities to this tool for more specific requirements.

Chapter 12, "Testing the AutomatedTest Tool," concludes the book by using the AutomatedTest tool to solve real-world software testing problems. This chapter also illustrates how to test the AutomatedTest project itself using methods introduced in this book.

About the Examples

The examples start with the programming precepts of C#. The goal is to use the predefined methods of a programming language to complete an AutomatedTest tool project. There are three kinds of example code in the chapters:

- Simple examples to demonstrate using C#
- Example projects to be tested by the AutomatedTest tool
- Sample code of the AutomatedTest tool project

The code examples in the first category most often appear in Chapters 3 through Chapter 6. Thereafter, Chapters 7 through 11 are totally dedicated to automating the test project. The sample code of these chapters is in the third category. There are only three examples of the second category, simulating a real assembly under test. They are implemented in Chapters 4, 8, and 11 immediately before the coding of the AutomatedTest tool begins. At the end of each chapter, one of the examples is submitted to the newly coded test tool.

In this book, the demonstration examples are managed assemblies. However, Chapter 5 briefly introduces interoperability between unmanaged and managed assemblies. When there is a need to test an unmanaged COM component, a simple conversion can turn an existing COM into a managed assembly, which can be tested by the AutomatedTest tool.

In Chapter 3 and thereafter, some code is added to the project in each chapter. At the end of each chapter, the sample code can be compiled to produce an executable assembly, and the testing tool achieves different degrees of automation until a fully automated test tool is developed by the end of the book.

The code in Chapter 4 enables the AutomatedTest tool to reflect an assembly under test. Chapter 5 implements XML documentation and an MS Excel worksheet for the testing case data stores. Chapter 6 introduces many fundamentals about CodeDom, which is used to generate test scripts automatically. Sample programs demonstrate how to use CodeDom to generate programs, but code is not added to the AutomatedTest tool in this chapter.

In Chapter 7, we'll resume coding the AutomatedTest tool and making this tool capable of generating a test script for a given assembly. After the addition of the code in Chapter 8, the tool will have the capability of testing parameters passed by objects and conduct integration testing. Chapter 9 covers the important test topics of verification, validation, and result presentation. After this chapter, you can use the tool to generate a fully functional test script. However, you still need to issue a line command to run and to deploy the test script. Chapter 10 guides you in adding the last batch of code (which builds and deploys the assembly from the test script) so that you achieve a fully automated test tool. At this point, the user can feed an assembly to the tool and get the test results with the classes, methods, and properties tested.

Finally, Chapter 11 discusses how to upgrade the tool with more capabilities. An example is given on how to add methods to generate code testing against the Windows Registry.

Where to Find the Source Code

The sample and project code for each chapter can be downloaded from `www.sybex.com` by searching this book using the title, the author, or the ISBN number, 4320. This saves the readers from having to type in the code. It also includes a complete compiled version of the project, which allows the readers to put the AutomatedTest into immediate practices for their software project.

To execute the `AutomatedTest.exe` file, you can copy it to a computer system. The minimum requirements for a computer system are as follows:

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

Although the sample code in this book is developed under Microsoft Visual Studio .NET 2003 Integrated Development Environment (IDE), there are other open source .NET IDEs available for free download:

Eclipsing .NET IBM released Eclipse .NET to the opensource community. This product works with Windows XP/2000/NT/98/95. You can download the components for Eclipse .NET at `www.eclipse.org`. After downloading eclipse-SDK-2.1.2-win32.zip, you install it in conjuction with Microsoft .NET SDK, which you can also download for free at `http://msdn.microsoft.com/netframework/technologyinfo/howtoget/default.aspx`. Then, get the open source C# plug-in through the Eclipse .NET IDE. (For more detailed information about downloading and installing these programs, go to `www.sys-con.com/webservices/articleprint.cfm?id=360`.)

`#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`.

DotGNU Portable .NET This open source tool includes a C# compiler, assembler and 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.southernstorm.com.au/portable_net.html`.

Chapter 1

Software Testing: An Overview

Any software product, no matter how accomplished by today's testing technologies, has bugs. Some bugs are detected and removed at the time of coding. Others are found and fixed during formal testing as software modules are integrated into a system. However, all software manufacturers know that bugs remain in the software and that some of them have to be fixed later (Beizer 1990). Testing is a necessary prerequisite for the successful implementation of a software product, but with the testing technologies currently available, it is often regarded as being difficult, tedious, time consuming, and inadequate. It is reported that bugs in the final products cost the United States economy $59.5 billion per year (Tassey 2002).

Testing accounts for 25% to 50% of the total budget in many software development projects. A test team usually consists of engineers that act as manual testers, tool users, and tool developers. Both the budget and the personnel are important because a product under development should be tested as thoroughly as possible.

Today, there are many testing tools on the market. However, they are considered to be fairly primitive with regard to the quality required (Tassey 2002). They can test software with some degrees of automation so that the test engineers can devote more time to solving problems in high-risk areas, but their automation is limited to simple reverse engineering and recording test script by mouse clicks or keystrokes. Test engineers expect more powerful and flexible testing tools with more automatic features to catch up with the rapid evolution of the software technology.

The goal of this book is to show you how to develop a testing tool, the AutomatedTest tool, to test a complex software product thoroughly with minimum human interaction. In addition to learning how to develop the testing tool, you'll gain the knowledge you need to implement the tool. For a controlled and gradual improvement of the automatic testing process, the discussion of the development of this tool contains comprehensive descriptions of Reflection, Code Document Object Model (CodeDom), late binding, Extensible Markup Language (XML), and MS Excel API programming. By the end of this book, you'll be able to use the tool to automatically test classes, methods, parameters, objects, and the Windows Registry based on the testing needs of a given assembly.

There are many computer operating systems and development languages serving today's software industry, such as Java, C++, Visual Basic, C#, Linux, Unix, and Windows. Some software engineers prefer one over the others, and some work with more than one.

NOTE Although this book's examples are written in C#, it does not imply an endorsement of Microsoft products or criticism of the other development environments and platforms. In fact, the methodology introduced in this book can be extended to other development environments and platforms. Java and .NET provide highly object-oriented environments to complete the testing tool projects. As you study the sample code in this book, you will learn the concepts of namespaces, data types, .NET Reflection, and .NET CodeDom with regard to software test automation.

In the .NET world, compiled physical applications, such as the files with `.exe` and `.dll` extensions, are often referred to as assemblies. An assembly could contain a single or multiple modules, such as namespaces and classes. Within such a hierarchy, fields, methods, properties, and events supported by the classes are the target of test units. This book covers the necessity, knowledge and techniques to complete the AutomatedTest project which at last will be capable of writing and executing code to test all the aspects of an assembly automatically. The discussion of the development of the AutomatedTest tool is based on automatically testing a real-world assembly. Example code for the tool is derived from actual test projects. They can be reused and referenced from tester to tester and from project to project. This tool offers features to conduct an investigation on the assembly under test by collecting testing information from the assembly and writing a test script to test it. . These testing tasks are accomplished dynamically and are not difficult, tedious, or time consuming to the test engineers after this development.

This chapter begins with a discussion of the basic software test concepts, the purpose of software test, the selection of a programming language and how they apply to automation. It also includes a discussion on how to effectively use these concepts to improve the current infrastructure of the software testing.

Purpose of Software Testing

Currently, total sales of software products have reached $180 billion a year in the United States. Due to the bug-bearing features, catastrophes caused by software products have happened from time to time.

In April 1999, a software defect caused the failure of the Cape Canaveral launch of a $1.2 billion military satellite. This has perhaps been the most expensive software failure in the history of the software industry. It subsequently triggered a complete military and industry review of the U.S. space launch programs, including software integration and testing processes.

You may still remember the loss of the NASA Mars Climate Orbiter in space in October 1999. That was due to the failure of a part of the software to translate English units of measurement into metric units. In 2002, I developed a module for an optical test instrument that validated there would be no mixture of such measurement units. If the Mars Climate Orbiter had developed such a module, the spacecraft would have been in orbit to this day.

This book is for everyone who is interested in software quality, including senior administrators, software project managers, software developers, software testers and software end users. When starting a new project, software organizations comply with development models. Software end users, developers, test engineers, and higher-level administrators are all involved in defining the requirements, specifications, and testing strategies. In most of the models, software

testing is planned at the beginning of the process and conducted parallel to the development life cycles. One cannot test a product before understanding it. Testing a new product under development is always a learning experience for the test engineers. The time and effort involved depends on the complexity of the product and the experience of the test engineers. To a great extent, one of the benefits of using an automated tool is to prevent the test engineers from spending too much time learning a new product. Engineers are then able to focus on more complicated and high-risk problems.

Software testing is the process of exercising an application to detect errors and to verify that it satisfies the specified requirements. During the software development life cycle, software developers and test engineers are working both to find bugs and to ensure product quality. The delivered software product must include all required functions and be compatible with the customers' hardware.

For a long time, software testing has been conducted manually; that is, a human tester runs the application using predefined processes. Since the beginning of the software industry, software engineers have made great efforts to automate the software testing process. Many successful companies have manufactured software test tools that are now on the market. Today, there are many commercial software tools that can be used to find bugs so that they can be fixed prior to the product release. As mentioned earlier, these tools possess some automation (performing reverse engineering and writing the test scripts), but the following deficiencies are often present:

- The test scripts often need debugging.
- None of them can complete an entire test process independently.
- They implement processes that may not be consistent with the organization's software design.
- The reverse engineering process is separated from the test script generation.

In many cases, generating or recording a test script for each member within an assembly is an exhaustive job for the test engineers. Making up and documenting the testing data with the current tools is a 100% purely manual task. Therefore, the capabilities of these tools with regard to automation are limited.

With the automated software testing tool developed in this book, testers do not need to write test script by hand or by recording test scenarios. The software testing process will work with the least amount of human interaction. This tool is designed to be reusable and meet the testing needs of most software products. In other words, you are going to learn how to make reusable modules to test other code.

Expectations of Automated Software Testing

With a fully automated tool, software testing can be expected to save time, observe standards, and enhance product quality.

Software testing has always been important for the software industry. It is usually common to train the testing engineers and have them learn the testing techniques at the beginning of a project. As the experience and skill levels increase in the team, the coverage of the automated software testing increases. Remember, testing a newly developed product is always a learning experience for a software tester. Deep knowledge of a project makes full automation possible.

Technically, many testing tasks can be automated. However, the decision of whether to automate the testing tasks for an organization is a management issue as well as a technical one. The analysis should include the following:

- *In today's society, software projects are complex and intend to solve complicated problems.* Commercial software testing tool manufacturers usually need time to learn about a particular problem and to catch up with the technologies. In order to meet the time frame for a project, a testing team should develop an automated testing tool that is complementary to the commercial testing tools.
- *Sometimes, test engineers need to determine whether to incorporate automated testing into existing projects or into a new one.* At the beginning, the testing process always involves manual testing. Test engineers use this as a learning process. As the project progresses, test engineers become more knowledgeable about the product and potential problems become more foreseeable. Automated testing tools then can be added to deal with the possible problems. Eventually, this tool greatly benefits the future projects in an organization.
- *Organizations employ the Extreme Programming (XP) practice for risk-prone projects with unstable requirements.* They don't emphasize detailed requirements documentation. Instead, their code is constantly being modified and updated. Test scripts are part of the source code control along with the program code. Thus, automated testing is critical to the XP practices so that the test scripts can be modified and rerun for each of the frequent development iterations.
- *Occasionally an organization may not be interested in the commercial software testing tools.* In order to manage quality assurance, developing an automated testing tool significantly enhances the development project.

The life cycle of the software development model also affects the extent of an automated testing process:

- *Usually, modification occurs frequently at the beginning of a project.* Your goal is to develop a tool to generate test scripts automatically, which would reflect the changes and significantly enhance testing efficiency with unstable products.

- *Commercially available testing tools are said to be capable of testing graphical user interface (GUI) components.* Most of the time, users need to record a series of mouse clicks and keystrokes. Testing data of the recorded test script are usually hard coded. Sometimes, the recorded test scripts need to be revised and debugged before they can perform the testing. Thus, using and reusing these tools on an unstable product would be a nightmare. An automated testing tool should be capable of recognizing and verifying the GUI components by itself and generating a data driven test script.
- *As the technologies advance, programming becomes easier.* The development life cycle becomes shorter, giving the testers less time to test the products. An automated software test tool should relieve some of that pressure so testers have more time to identify high-risk areas of the project.

Automated Testing and XP Practice

Kent Beck created Extreme Programming (XP) in his book *Extreme Programming Explained: Embrace Change* (Addison-Wesley, 1999). By his definition, XP is a lightweight technique that focuses on coding for a risk-prone software project. It avoids the detailed specification and regards every task as a simple one. In order to accomplish a complex problem, it relies on frequent iterations and communications between developers, testers, and customers.

The first priority of managing an XP project is to make sure that in the end, all the needs of the customer are satisfied. However, XP members believe that it is not necessary for customers to foresee the needs of a product. Software developers are not fortunetellers and do not need to specify all requirements before coding. It is more useful to provide customers with a simple workable product as early as possible. Then, customers can explore the product and add or change their requirements. Developers will eventually achieve the goal through constantly revising the code, automatic testing, and continuous integration of the product.

Automated testing is a key to the success of using XP practice. Tests are needed when new code is added and when a method or a line of code is modified or deleted. The code is usually under constant change and evolving, and sometimes developers aren't aware of all the changes taking place. Only through frequent testing can programmers realize that the changes of the code are validated.

Due to constant changes in requirements, specifications, and the code, the tests should be automated so that none of the changes break the system. Unlike the code created in other development models, the XP code is in a liquid state. It can be redesigned, refactored, deleted, and completely redone. Later, the tests will make sure that the system still works. XP needs an iterative testing for the public interface of classes and components. The implementation of the system is under drastic change while the automated tests validate that the system still fulfills the contract of the interfaces. Only after a new or modified feature is validated by tests can it be

integrated. Everything that can potentially break must be tested. The AutomatedTest project in this book will help XP teams automate the testing.

Software Test Engineers

This book focuses on the development of a tool for automated software testing. The implementation incorporates the tester's attitude of "test to break." As an automated tool, it is able to "use" a product from the perspective of an end user, has the "desire" for quality, and has "attention" in details. It is a user-friendly tool for both technical and nontechnical personnel. Many times, a tester is considered to be a good programmer also. This tool is capable of writing test scripts as well as an experienced tester who has a deep understanding of the software development process. Eventually, it reduces the learning curve so that the testers can focus on high-risk areas. If your team employs the XP practice, you will find this AutomatedTest tool extremely useful.

Today, it is common that testers and developers work closely together for one project. One testing tool may not be sufficient for solving all the problems that may appear. If you are a project manager, it will be more efficient for you to have both manual testers and automated testers on your team.

With effective communication, the automated and manual testing sections will complement each other and enhance each other's capabilities. Thus, a manager may want team members who are skillful in both manual and automated testing techniques. A quality product will benefit from a testing team with members who have different kinds of testing experiences. For example, some may have experience with the tools available on the market. Some may be good at using manual testing to test more complicated and high-risk areas of a product. Others might have significant programming experiences using C#, Visual Basic, C/C++, Java, or other languages to develop tools.

How to Automate Software Testing

The software industry has seen many great works on software testing. Textbooks have documented how to make testing plans and testing strategies and how to conduct automated software testing with the tools that are available on the market. Test engineers have benefited a lot from the accumulated knowledge, and software organizations have encouraged their testing teams to observe these documents.

The high-level development languages, such as Java, C#, and Visual Basic .NET, are object-oriented languages that aim at enabling programmers to quickly build a wide range of applications. The goal of these languages is to shorten the software development life cycles by providing the Common Language Runtime (CLR), the Java Virtual Machine (JVM), cross-language

and cross-platform support, and automatic memory management. They also handle other low-level plumbing issues, such as type safety, available low-level libraries, array bounds checking, and so on. Thus, developers actually spend their time and resources on their application and business logic. Traditional testing methods cannot meet the speed of technological advancement. To meet the new time frame, there must be better and faster ways to test software products. Automated testing is receiving a lot of attention due to technological advancement as well as the ever increasing complexity of software products.

A .NET assembly organizes a software product in the structure of namespaces, class types, and class members. Assemblies are built by the Microsoft intermediate language assembler (MSIL). Then, the Microsoft disassembler can reverse-engineer the assemblies. We'll take advantage of.NET to develop a tool that should be capable of recognizing the classes and members in an assembly (a DLL or EXE file) and automatically write a test script for testing each method of this component. Thus, the automated software test tool developed in this book is capable of the following:

- Learning the assembly under test dynamically
- Conducting tedious or repetitive tasks dynamically
- Generating test scripts and running the scripts in batches in a scheduled manner
- Testing interfaces of COM objects and other software components with a specified data set
- Accessing a database to verify the testing results
- Accessing the Windows Registry to verify the testing results.

According to the developers of Microsoft Visual Studio .NET, the .NET Reflection namespace is able to reflect a given assembly. The .NET CodeDom namespace can write the test script dynamically. Late binding can invoke a method. The test data and test results will be presented using XML and MS Excel worksheets. Thus, making use of the .NET Reflection, .NET CodeDom, XML programming, and MS Excel API programming, the automated testing process can be viewed with six steps as shown in Figure 1.1. By the definition of regression testing, code is under constant testing and under constant change during the development life cycle. When bugs are detected and fixed, the regression testing will confirm that the fix is accurate and doesn't cause side effects. Thus, the iteration in Figure 1.1 is accomplished by regression testing.

Think of the relationship between the .NET Reflection namespace and the assembly under test as being similar to the relationship between a prism and sunlight. A prism reflects the sunlight into its spectrum of wavelengths; similarly, the .NET Reflection namespace disassembles an assembly into class, property, and method members. Thus, it automatically accomplishes the information gathering process. This process is similar to test engineers spending time learning the product. Thanks to the Reflection namespace, the test engineers can devote their time to identifying other risks.

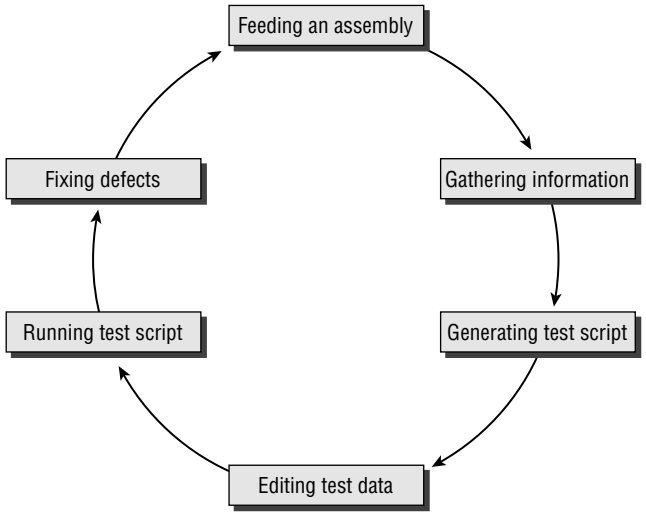

FIGURE 1.1
Six steps of the automated software testing

After the information is gathered, the .NET CodeDom, which is a new area of technology to software developers, becomes involved. Chapter 6 in this book is devoted to discussing the .NET CodeDom features in detail. Ultimately, the development of the AutomatedTest tool employs this namespace to write the test script based on the information disassembled by the .NET Reflection namespace.

Software test engineers have used tools on the market that are capable of reverse-engineering a software product. However, it sometimes becomes tedious to instruct the tools to write a test script for each of the members within an assembly. It is especially true when the assembly under test has hundreds of members. With the combination of the .NET Reflection and the .NET CodeDom, the AutomatedTest tool combines the information gathering and script writing processes with minimum interaction from a human tester.

To enable the data editing capability, the testing tool developed in this book can use an XML document or an MS Excel worksheet for data stores. XML has been widely accepted by many industrial standards. Enabling XML data storing enhances the software test automation. It also can be applied across different development environments and platforms.

The Excel worksheet is used to store testing data because of its popularity across the MS Windows platforms. The programmable MS Excel API offers great flexibility for testers to organize the test data. Then, the same MS Excel API programming techniques are incorporated into writing the test script, which tests the assembly against the testing cases and posts the test results.

In the Microsoft Visual Studio .NET integrated development environment (IDE), assemblies are built using the MSIL assembler. In order to run the test script, the test engineer can programmatically invoke the test with *late binding*—an ability to discover underlying types, methods, and properties of a given object and invoke the methods at runtime. Employing the late binding technique makes the automated testing tool more efficient. Therefore, you obtain a fully automated test tool with the assistance of the MS .NET programming environment.

Commercially available tools are not capable of testing a product thoroughly. When a new testing feature is requested, a testing tool manufacturer might promise that the tool would include this feature in the next release. To cope with the diversity of software, the tool developed in this book has enough "brain cells" to learn new lessons. By the end of this book, you will be able to upgrade this tool programmatically.

Test engineers have experienced the features of capturing screens and recording scenarios by using the tools on the market, which conduct unit tests, advanced stress tests, and load capability tests. The AutomatedTest tool doesn't require reverse engineering or recording actions. In its earlier stages, you can use this tool as a complement to the available testing tools and to the test engineers' skills and experiences. However, as the AutomatedTest tool project progresses, the tool will eventually address as many testing issues as possible.

Therefore, by using this fully automated testing tool, test engineers will be relieved from reverse-engineering, writing tedious scripts, and creating data stores. They can spend more time on tackling the high-risk areas of software. Furthermore, programming in the MS .NET environment provides test engineers with the advantage of developing a tool within the organization's time frame and budget. This applies to many other development environments and languages, such as Eclipsing .NET, DotGNU Portable .NET, Java, Visual Basic 6.0, C++, C# and Visual Basic .NET.

Software Testing and Programming Languages

It has already been mentioned that there are a lot of testing tools available on the market, and testing organizations have used the tools and achieved great success. Some popular testing tools write test scripts in Visual Basic 6.0 and execute the script in Visual Studio IDE. There is also literature that introduces how to write test scripts in Visual Basic 6.0 by hand. Some other tools are written in Java, such as JUnit, and JProbe. And there are tools to write test scripts for software products run on Unix, Linux, and other platforms. The sample code of this book will use C# of the Microsoft Visual Studio .NET environment as the programming language for both the tool and the test script.

The automation targets are mostly focused on problems from certain well-defined areas. That is because these problems and areas have experienced software failures in the past. Testing tools often write test scripts with a few repetitive code patterns to test different methods within a class.

For example, Listings 1.1–1.4 are excerpts from a test script generated in Chapter 8. These examples will help you understand how to test a few commonly encountered testing situations programmatically. At this point, compare the difference between these listings rather than trying to start a project using them.

Listing 1.1 Code to Test the *objAdvancedCalc.Square()* Method

```
shtRow = 12;
mName = "Square";
range = xSheet.get_Range("C12", "C12");
double length_12 = double.Parse(range.Value2.ToString());

try
{
  xResult.Cells.set_Item(12, 4, objAdvancedCalc.Square(length_12));
  TestPass(xResult, shtRow, mName);
}
catch (System.Exception err)
{
  TestFail(xResult, shtRow, mName, err.Message);
}

xResult.Cells.set_Item(shtRow, 6, "length = " + length_12);
range = xSheet.get_Range("D12", "D12");

if (range.Value2 != null)
{
  xResult.Cells.set_Item(shtRow, 5, range.Value2.ToString());
}
```


Listing 1.2 Code to Test the *objAdvancedCalc.SimpleCalc()* Method

```
shtRow = 14;
mName = "SimpleCalc";
range = xSheet.get_Range("C14", "C14");

Assembly appDomain_14 = Assembly.LoadFrom
➥(@"C:\Temp\LowLevelObj\Bin\Debug\TestLowLevelObj.dll");

Type TestClass_14 =
➥appDomain_14.GetType("LowLevelObjTest.TestLowLevelObj");

object objInst_14 = Activator.CreateInstance(TestClass_14);
MethodInfo mi_14 = TestClass_14.GetMethod("StartTest");

LowLevelObj.SimpleMath lvlMath_14 =
➥(LowLevelObj.SimpleMath) mi_14.Invoke(objInst_14, null);

range = xSheet.get_Range("D14", "D14");
```

```
LowLevelObj.Math_ENUM operation_14 = LowLevelObj.Math_ENUM.Add;

range = xSheet.get_Range("E14", "E14");
int Result_14 = int.Parse(range.Value2.ToString());

try
{
  objAdvancedCalc.SimpleCalc(lvlMath_14, operation_14, ref Result_14);
  TestPass(xResult, shtRow, mName);
}
catch (System.Exception err)
{
  TestFail(xResult, shtRow, mName, err.Message);
}

xResult.Cells.set_Item(shtRow, 6, "lvlMath = " + lvlMath_14);
xResult.Cells.set_Item(shtRow, 7, "operation = " + operation_14);
xResult.Cells.set_Item(shtRow, 8, "Result = " + Result_14);
```

Listing 1.3 Code to Test the *objAdvancedCalc.Sum()* Method

```
shtRow = 15;
mName = "Sum";
range = xSheet.get_Range("C15", "C15");

double[] anArray_15 = new double[range.Value2.ToString().Split(',').Length];
foreach (string z in range.Value2.ToString().Split(','))
{
  anArray_15[tempArrayIndex] = double.Parse(z); tempArrayIndex++;
};

tempArrayIndex = 0;

try
{
  xResult.Cells.set_Item(15, 4, objAdvancedCalc.Sum(anArray_15));
  TestPass(xResult, shtRow, mName);
}
catch (System.Exception err)
{
  TestFail(xResult, shtRow, mName, err.Message);
}

xResult.Cells.set_Item(shtRow, 6, "anArray = " + anArray_15);

range = xSheet.get_Range("D15", "D15");

if (range.Value2 != null)
{
  xResult.Cells.set_Item(shtRow, 5, range.Value2.ToString());
}
```

Listing 1.4 Code to Test the *objAdvancedCalc.GetType()* Method

```
shtRow = 16;
mName = "GetType";

try
{
  xResult.Cells.set_Item(16, 4, objAdvancedCalc.GetType());
  TestPass(xResult, shtRow, mName);
}
catch (System.Exception err)
{
  TestFail(xResult, shtRow, mName, err.Message);
}

range = xSheet.get_Range("C16", "C16");
if (range.Value2 != null)
{
  xResult.Cells.set_Item(shtRow, 5, range.Value2.ToString());
}
```

The four segments of code test different members of an assembly and solve different requirements. In general, they repeat a similar coding pattern approximately as follows:

1. Set up the necessary parameter(s)
2. Execute the method under test and catch the errors
3. Present the result of the execution

The methods under test behave differently. Methods tested in Listings 1.1, 1.2, and 1.3 all take parameters. The code segment in Listing 1.1, test method `objAdvancedCalc.Square()`, passes the parameter by value. The code in Listing 1.2, testing method `objAdvancedCalc.SimpleCalc()`, passes the first parameter by object, which is usually done by stubbing or by using the mock-objects method with other testing tools. However, this segment reuses an existing test script, which is introduced in Chapter 8. The advantage of reusing a test script is obvious. The code in Listing 1.3 tests a method, `objAdvancedCalc.Sum()`, by passing an array parameter. The code is totally different from that in the other listings in order to handle the items in an array. The code segment in Listing 1.4, testing method `objAdvancedCalc.GetType()`, takes no parameter, which is the simplest case. Each of the code segments in Listings 1.1, 1.3, and 1.4 returns a value. Testers are interested in seeing whether the actual return value matches the desired return value. Thus, the resulting presentation will display the return values after the execution. The method under test in Listing 1.2 does not return a value. One of the parameters is passed by reference, and the status of this parameter is altered during the execution. The presentation has to inform testers of these changes.

These are a few examples of different testing scenarios. An automated testing tool will be implemented to test all aspects of a real-world software product.

It might seem easy to code this pattern manually when there are not many members under test. But it is time consuming to code this way for an assembly with a large collection of classes and members. Also, it takes human testers longer to gain insight on each method in order to code with accuracy. On the other hand, it is hard for a tool to code the subtle difference between methods the first time. Debugging is always needed for test scripts written by a human. With the AutomatedTest tool, the test process will become fast for the repetitive jobs and debugging will not be needed.

We all know the boundary condition for testing cases, but we often spend a lot of time implementing it to compose the testing cases. This book introduces a couple of automated methods for composing the testing cases with ease, saving the testers time to investigate high-risk areas and to spend more time on other manual testing jobs. As you update the automation capacity for your institution, the solution for problems that arise repeatedly will eventually be automated. Only new problems will be tested manually and become automated later.

No matter what language they use to write test scripts, testers must have knowledge in testing and have skills in programming. The methods illustrated in this book can be extended to the languages and platforms you prefer.

In fact, C# was created for software developing, not for software testing. It is understandable that some C# experts use third-party software testing tools for their software projects. You may wonder how a programming language could ever be used to develop a testing tool. This is the area that will be explored in this book.

In the past, Visual Basic and Java have been used more often than other languages to write test scripts. For example, Rational's testing tool uses a reverse-engineering method and generates scripts in Visual Basic. The book *Visual Basic for Testers* (Sweeney 2001) introduces some techniques of writing test script, but it doesn't cover techniques for script automation. Java developers use JUnit as a testing tool. Using a programming language such as C# as a testing tool is a brand-new concept to many readers. This book will not only teach you how to write a test script for a particular assembly, it will also teach you how to develop a general tool to write test script dynamically. Thereafter, different organizations and different functional software projects can employ this method for their software development. To recap, the method will focus on solving the following problems:

- How to use predefined functions of a language (in this case, C#) that return relevant information about the assemblies under test
- How to create a front end with basic controls to view test information and results
- How to make and store testing cases in a database (in this case, MS Excel and XML documents) to drive the automatic generation of a test script.

- How to create a presentation showing the information of a component under test (in this case, using an Excel worksheet)
- How to implement a mechanism to generate test scripts for testing a class, a property, and a method (in this case, using the .NET CodeDom).
- How to enable the test scripts to test parameters passed by value, by references, and by objects
- How to enable the test scripts to test against the Windows system registry
- How to present the test results so that the developers will have as much information as possible to fix a defect responsible for the test failure

Using C# for Automation

The previous sections have discussed the possibilities of automated testing. This book is also an evolutionary result of software test techniques. Not all types of tests can be automated, so test engineers will still be necessary. But this book attempts to broaden the area of test automation by giving test engineers the knowledge to use automation techniques. As more test areas are automated, software test engineers will have more time to plan their test strategies and manage their budget and other resources efficiently. Test areas will be planned with more confidence and defects fixed on schedule. Especially, the high-risk areas will be tested manually and adequately. Developers, testers, and users will find new confidence in the production of products whose quality is dramatically improved.

Currently available testing tools may lack the capability to test the products developed with .NET platform and C#. Test engineers with desire for automation have to develop their own tools to accomplish their tasks.

Engineers know that C# is an advanced programming language. Although it is not designed to be a testing tool, it happens to have many methods that can be used in the testing process.

For example the .NET Reflection namespace can reflect an assembly as a prism reflects the sunlight. The .NET CodeDOM namespace can write code for the script. On the other hand, C# is a completely object-oriented programming language. The tool developed with C# will be easily reusable, portable, and maintainable. The COM interoperability enables the function to test COM/DCOM developed in other development environments.

However, when the test engineers need to use C# features to present the test results, a new form of the presentation may seem unfamiliar to the end users. In this case, using a broadly accepted method will enhance the usability of the new test method. With advanced programming techniques, one is able to reach the goal. For example, this book shows you how to program an XML mechanism and an MS Excel worksheet as the visual presentation of the automated test results. At first, an MS Excel worksheet presents the information of a component

under test and provides space to store testing data. Later, the MS Excel API programmed in the test script presents the test result, provides references for the regression testing, and gives information for correcting the defects.

Test Scripts

A test script simulates an end user to run the application using a prescribed scenario so that a software product can be tested automatically. Traditionally, an automatic test script is written by a tester to test the methods of an assembly. When the test engineers write the automated test scripts, they must take time to define and analyze the requirements. As one or more of these values change during the software development process, the testers have to revise or rewrite the test scripts. This process will be repeated until the product is stable. Then the scripts may be interactively run to perform unit testing, integration testing, or regression testing. During the whole process, the task of the testing team is to improve and debug the test scripts and to meet the ever-changing testing requirements.

To reduce the tedious and time-consuming manual script-writing process, the automated testing tool is capable of detecting changes to the software definition automatically. If changes are made to the definition of the software, the tool will write a new test script accordingly. If the specification of the software is stable throughout the development life cycle, the test script can be rerun throughout the development life cycle for unit, integration, and regression testing.

The rules for developing a good software product apply to developing a good test script. In order to generate an efficient test script, the tool should have the mechanisms to reflect the detailed information of the project under test. A test script generated by the tool should possess the following features:

Reusability It requires great efforts to develop a testing tool. Software engineers will be more willing to make these efforts if they can meet the requirements of one project and reuse the tool for future projects. Because this book follows the management of a conventional software development to automatically generate reusable test scripts, it uses C# as a highly object-oriented language. The reusable tool and test script explore an assembly under test through namespaces, classes, methods, and properties. A test script for one testing case can be used to test another testing case. Finally, the tool itself can be reused and updated from project to project.

Readability Standard naming conventions must be used for variables and constants in the generated test script so the tester can review the code. The test script will precisely test the features of a software assembly. If there are some special needs for a software product, you can choose to update the AutomatedTest tool or revise the generated scripts. For example, if these special needs happen frequently, it is worthwhile to update the AutomatedTest tool. It makes software testing easier for the future projects. If the new requirements are not

typical, the test engineer may decide not to update the tool but instead to modify the test script. In this case, a readable test script will be easier to modify.

Maintainability It is common knowledge that different software organizations use different software engineering practices. Requirements and designs are different from product to product, although they may share many common features. There are enough features in the AutomatedTest tool to cover most software products. When new features are expected in future projects, the AutomatedTest tool can be updated. An example of updating this tool to test the Windows Registry is presented in Chapter 11.

Portability Test engineers have the motivation to develop tools that can be used by their colleagues. With a simple installation process, the AutomatedTest tool is capable of performing the required testing tasks on different computer systems. Thus, other testers—as well as developers and IT and product learning personnel—can run the testing tool to detect bugs and defects.

As you work through the book, keep in mind that all successful software project rules apply to the development of the testing tool. No engineer writes perfect code. A test engineer is passionate about finding defects in code written by the others. This testing tool is able to generate an executable test script without debugging, which is a big time-saver. However, one must allow enough time to debug and test the AutomatedTest tool during and even after the tool development. Otherwise, it may introduce its own bugs into the software under test.

Summary

Recent findings by scientists and economists suggest that the infrastructure of current software testing is inadequate . Today, testers are encouraged to develop testing tools for an organization. This book introduces some new methods that are used to develop an automated testing tool. The methods and tool development will rectify the deficiencies of current testing tools as follows:

- You can use the methods introduced in this book to develop a testing tool in your preferred language and to develop one that fits the culture and business practices of your organization.
- The discussion of the methods and the tool is general and independent of other testing tools. You can add your own code to make this tool generate test scripts that compatible and interoperable with your current testing tools and methods.
- You can include this testing method in the source code control. Thus, the development and test team can update and share the testing cases and scripts easily.

- This tool will generate one test script to conduct a thorough test for an assembly or a class. It looks for what to test by itself. Therefore, testers can devote their time to exploring the high-risk areas of the products.
- A fully automated testing tool requires less time to learn and solves complex problems effectively. Thus, personnel from different departments can be involved in testing and find as many defects as possible at the early stages of development.
- The automation this tool provides eliminates the need for reverse engineering and a capture/replay procedure to write a test script. An automatically generated data store drives the test script generation. One script completes thorough unit testing for an assembly.
- The integration testing uses complete and real objects from the thorough assembly tests. Thus, it avoids the pitfalls of stub and mock objects.
- You can always upgrade this tool with new capabilities to keep up with the testing challenges in your organization.

It is important to observe software development management practices in order to successfully develop a testing tool. Chapter 2 will briefly review the current test tools that are available on the market. Starting from Chapter 3, the context of the book will include discussions of the technical fundamentals for software test and tool development. When there is enough background knowledge, code will be added to the AutomatedTest project in each chapter. In addition, three other sample projects will be developed in chapters 4, 8 and 11 to test and demonstrate the capabilities of the tool under development.

Chapter 2

Current Testing Infrastructure vs. the Proposed Testing Methods

Software has become an intrinsic part of business over the last decade. Virtually every business in every sector in the United States depends on it to aid in the development, production, marketing, and support of its products and services. By 2000, total sales of software reached $180 billion annually. The number of software engineers and programmers is approximately 1,282,000. Reducing the cost of software development and improving software quality are important objectives of the U.S. software industry.

These figures are based on a study by Dr. Gregory Tassey conducted under the auspices of the Research Triangle Institute and the National Institute of Standards and Technology (Tassey 2003). The same study also reported that the software industry suffered an economic loss because of an inadequate infrastructure for software testing. Software failures cause about $59.5 billion worth of loss to the economy annually, which accounts for almost one percent of the nation's gross domestic product. New testing tools and methods are needed immediately. According to his estimate, reducing one-third of the bugs in today's software will save the nation's economy $22.2 billion.

There is a wide variety of software testing tools on today's market. The manufacturers of these tools include Rational Software from IBM, Mercury Interactives, Segue Software. There are tools from other vendors and also some open source test tools such as Ant, JUnit, and JProbe.

As a software test engineer, I have appreciated the great capabilities of these testing tools, although they are not as automatic and effective as they could be. This chapter includes a brief discussion of some of their features. The common limitations of commercial tools include that they fail to keep up with the technology advancements and are hard to adapt to new design processes in the software industry. In this chapter, I'll propose methods for an improved test infrastructure to overcome these limitations.

Let's discuss some of the tools and their features for software test automation first.

Types of Software Testing

Manufacturers have developed tools to cover a wide range of testing tasks. However, utilizing these tools will not address the increasing complexities of software applications. Today, software organizations are incorporating multitier architectures, object-oriented design and programming, different kinds of interfaces, client-server and distributed applications, data communications, and huge relational databases, which all contribute to the exponential growth in software complexity. Test tool developers are unable to duplicate these complexities before a project progresses, making it often difficult or impossible for them to resolve the problems they present in advance. In addition, innovation from software engineers continues to improve the technology and add complexity to software projects. The tool developers can not predict

these improvements, and so they must be considered on an individual basis to determine how to test.

For these reasons, it's imperative that each software organization develops its own tool to solve its unique testing issues. In order to assure quality products within an aggressive time schedule, the organization must develop an effective test approach with a scope in focus.

Determining a testing scope is critical to the success of a software project. Due to the short time frame and the intensive human interactions associated with the available software testing tools, it can become difficult to test all components of an application under development. It is always desirable to test the software thoroughly, but that's not always possible due to time and budget constraints. If the testing tools are capable of carrying out testing jobs and composing testing cases with full automation, human testers will have enough time to focus on high-risk areas.

Without a fully automated testing tool, test engineers usually determine critical requirements and high-risk areas by analyzing the requirements that are most important to the customers and the areas within the application that receive the most user focus. Decisions are often made to do no testing at all in noncritical areas, which can cause problems and even disasters in the future and turn these noncritical areas into critical ones. Thus, the scope of a fully automated testing tool should be every line of the code, and the code should be tested continuously day and night until most of the defects are found and the product is released.

After the testing scope is determined, you usually need to plan the testing phases, including the types and timing of tests to be performed in both the pre- and postrelease stages. Commercially available software testing tools are used to perform the following types of testing in a complete cycle:

Unit testing This type of testing tests individual application objects or methods in an isolated environment. It verifies the smallest unit of the application to ensure the correct structure and the defined operations. Unit testing is the most efficient and effective means to detect defects or bugs. The testing tools are capable of creating unit test scripts.

Integration testing This testing is to evaluate proper functioning of the integrated modules (objects, methods) that make up a subsystem. The focus of integration testing is on cross-functional tests rather than on unit tests within one module. Available testing tools usually provide gateways to create stubs and mock objects for this test.

System testing System testing should be executed as soon as an integrated set of modules has been assembled to form the application. System testing verifies the product by testing the application in the integrated system environment.

Regression testing Regression testing ensures that code modification, bug correction, and any postproduction activities have not introduced any additional bugs into the previously tested code. This test often reuses the test scripts created for unit and integration testing. Software testing tools offer harnesses to manage these test scripts and schedule the regression testing.

Usability testing Usability testing ensures that the presentation, data flow, and general ergonomics of the application meet the requirements of the intended users. This testing phase is critical to attract and keep customers. Usually, manual testing methods are inevitable for this purpose.

Stress testing Stress testing makes sure that the features of the software and hardware continue to function correctly under a predesigned set and volume of test scenarios. The purpose of stress testing is to ensure that the system can hold and operate efficiently under different load conditions. Thus, the possible hardware platforms, operating systems, and other applications used by the customers should be considered for this testing phase.

Performance testing Performance testing measures the response times of the systems to complete a task and the efficiency of the algorithms under varied conditions. Therefore, performance testing also takes into consideration the possible hardware platforms, operating systems, and other applications used by the customers.

Automated Testing Tools in the Marketplace

Based on the priorities set in the scoping phase, engineers need to determine which set of test types and which testing tools to use. To select the proper testing tool, keep in mind that, by nature, humans use comparison as an evaluation technique. Testers compare the capabilities and effectiveness of different tools. Sometimes they may decide to develop their own tool or test manually. The minimum requirement of a testing tool is the capability for unit and integration testing. In the following sections, I'll discuss without bias some of the sources for testing tools. The list of tool vendors is incomplete, is purely for reference, and is not an implied endorsement of any company or product. The discussion describes the solutions offered by some of the tool vendors (most of the descriptions are based on their advertisements and web pages).

DevPartner Studio from Compuware Corporation

Compuware test tool developers intend to make their tools automatically detect, diagnose, and facilitate resolution of software errors and performance problems. The NuMega DevPartner Studio 6.1 includes the following components:

BoundsChecker Tests for memory and resource leak detection of Visual C++ applications.

JCheck Used for thread and event analysis in Java applications.

SoftIce Used for debugging application problems within the Windows environment.

CodeReview Checks Visual Basic code for known coding problems.

SmartCheck Provides improved error handling within VB applications.

TrueTime Provides performance analysis of the applications.

FailSafe Tests for debugging errors.

NuMega TrueCoverage Reports how much code your test script executes for products written in Visual Basic and Visual C++ and for Java applications, which provides insight into whether the tests are executing the different portions of the program's code. This test ensures that the test script actually tests the code needing to be tested.

Please refer to `www.numega.com/products/vc/vc.html` for more information.

Insure++ from Parasoft

Insure++ is an automatic testing tool for runtime error detection for C/C++. Sometimes errors in C/C++ code aren't evident during testing, and sometimes problems such as memory corruption may cause code to crash on one machine and run perfectly on another. Insure++ tries to reveal the hidden defects in the code under test. It was specifically designed to help developers and QA personnel find and fix the difficult runtime errors that other testing techniques fail to uncover. More information about Insure++ can be viewed through `www.parasoft.com/insure/home.htm`

Mercury Interactive

Mercury Interactive provides products to test custom, packaged, and Internet applications. The suite of tools includes the following components:

TestDirector This tool provides test planning, execution, and defect-tracking functionalities. It can organize tests in hierarchical folders to help you plan and track them. The tool can track both manual and automated tests. It allows user-defined fields to be added, and it can be used with a variety of database management systems (Oracle, Sybase, MS SQL Server, and MS Access). TestDirector integrates with Mercury's capture/playback tool (WinRunner) and volume/stress tool (LoadRunner). It provides a good set of standardized reports but few custom reports features. This is a Windows-based tool.

WinRunner This is a Mercury's capture/playback tool for Windows applications, Web applications, and applications accessed through a terminal emulation (e.g., 3270, 5250). It is context sensitive when recording (i.e., objects can move and the tool can still find them, meaning it is not always hard-coded). The tool uses a proprietary scripting language and