

**SYBEX Sample Chapter**

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

**Kanglin Li and Mengqi Wu**

## **Chapter 2: Available GUI Testing Tools vs. the Proposed Tool**

Copyright © 2004 SYBEX Inc., 1151 Marina Village Parkway, Alameda, CA 94501. World rights reserved. 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.

ISBN: 0-7821-4351-2

SYBEX and the SYBEX logo are either registered trademarks or trademarks of SYBEX Inc. in the USA and 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. Copyrights and trademarks of all products and services listed or described herein are property of their respective owners and companies. All rules and laws pertaining to said copyrights and trademarks are inferred.

This document may contain images, text, trademarks, logos, and/or other material owned by third parties. All rights reserved. Such material may not be copied, distributed, transmitted, or stored without the express, prior, written consent of the owner.

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 manufacturers. 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.

Sybex Inc.  
1151 Marina Village Parkway  
Alameda, CA 94501  
U.S.A.  
Phone: 510-523-8233  
[www.sybex.com](http://www.sybex.com)

**T**esting GUI components is different from testing non-GUI components. To automatically test non-GUI components, engineers usually develop test scripts to invoke members from an application under test and catch the return values of the invocations. Scripts can be developed using late binding and other readily available functions.

Today, almost all software applications are operated through a graphical user interface (GUI). An automated GUI test tool should be able to recognize all the GUI components. Then it needs to generate a test script to perform a set of mouse and keyboard actions to test the functionality of the underlying modules of the GUI. After the desired functions are performed, the tool needs to verify that the visible GUI representations are consistent with the intended functions and make sense to the end users.

Tool manufacturers have studied and developed various kinds of GUI testing tools. These tools have been designed to help test engineers record test scripts. However, none of the commercial tools have the capability of automatically generating GUI test scripts that perform and verify GUI actions and find bugs effectively. Engineers and economists have discovered their inadequacies and reported the economic losses caused by undiscovered bugs in software. This chapter will compare some of the popular tools. The discussion will be brief and aim to introduce the technologies and fundamental features of the current testing tools, which will ultimately help in developing the improved testing infrastructure.

## **Current GUI Testing Infrastructures**

Most of the GUI test tools use the popular capture/playback method as an easy way to record test scripts. The recorded test script then plays back the low-level mouse drags and keystrokes. Some tools can recognize certain GUI components as objects. Others record the coordinates of the mouse pointer actions.

Test engineers have observed that the currently available tools are not able to write robust test scripts to directly complete the software testing. They often spend their time manipulating the tools and later editing the recorded test scripts. If the test scripts fail to execute, they have to debug them. It turns out that tool users don't have enough time to compose effective test cases and execute the test scripts to find bugs.

### **Capture/Playback Is Not Automatic**

Tool vendors have long claimed that the test scripts recorded by capture/playback are automated GUI testing tools. But in reality, there are lots of pitfalls that impair the effectiveness of the test script generation and the test execution. During a test session, a capture/playback tool records all manual user interactions on the test object. In addition to performing operations on the application under test, the tool users are continuously interrupted to insert verification

points. Both the capture/playback and the process of inserting verification points are labor intensive and tedious. Then a manual programming process is needed to enter test data and other checkpoints.

After a test script is captured, tests are replayed and thus in principle can be repeated at any time. If the application under test behaves differently during a repeat test or if checkpoints are violated, the test fails. The test script records this as a defect in the application.

It is obvious that the capture/playback relies on trained testers to manually use the application under test. Without editing, the recorded script performs the manual operations in the exact sequence they were entered. In order to complete the desired testing, the recorded scripts need to be edited and debugged. Testers also need to determine and enter the test cases based on the manual testing experience. In many cases, if the system functionality changes, the capture/playback needs to be completely rerun. Otherwise, it requires engineers to edit and maintain the recorded scripts. This only reduces some of the efforts of the completely manual test script development, but it doesn't result in significant savings for an organization, and the automation is minimal.

Another nuisance is that a capture/playback can not be continued when a bug occurs. Re-execution of the capture/playback is needed after the bug is fixed. This scenario happens repeatedly. When the test script is finally captured, the existing bugs have been found during the iterative manual recording processes. The capability of finding bugs by playing back the test script is limited. It only helps with the regression testing.

Many tool vendors tell testers that their tools generate data-driven test scripts, when in fact they use a wizard to prompt the users to enter and store data in an external file. The script is generated by the capture/playback approach rather than by data entry prompted by the wizard. Although re-execution of the test script is driven by the external files, the script generation is independent from the testing data. Thus, during the capture/playback process, the tool users need to operate the application under test, enter verification points, and interact with the data wizard. Less-effective capturing tools even record hard-coded data in the test scripts.

The other often claimed feature by the tool vendors is the capability of comparing bitmaps. However, experts have warned that bitmap comparisons are not reliable. The problem is that even if one pixel changes in the application, the bitmap comparison reports a test failure. Most of the time, this one pixel difference is desirable and is not a bug. Again, if you manually modify the test script, this problem can be avoided.

However, testers have found that the capture/playback functionality can be useful in some ways. When creating small modular scripts, capture/playback provides an easy and fast way to capture the first test. Then testers can go back to shorten and modify the test scripts. Although maintaining the captured test scripts is difficult, some testers have used this method to create quick automated tests. Later, these test scripts are discarded or recaptured. Other testers have

used capture/playback to record test scripts by working on a prototype or a mock-up system at design time. Because the real functions of the application are not implemented at this point, there will be no bugs to interrupt the test script capturing. Editing and maintaining are continued thereafter.

Since this book proposes a fully automated method for data creation and script generation, you can use this method independently or adjunct to your existing testing infrastructure and avoid the manually operated capture/playback process. Testers can focus on generating test cases and running the test against as many cases as possible instead of focusing on recording, editing, and debugging test scripts.

## Test Monkeys

Software developers and users are all aware that it is inevitable to have bugs in software products, and engineers have used many testing methods to find them. Bugs are found during manual testing based on the test engineers' desire to deliver a high-quality product. Verification is conducted while the testers are operating the applications and comparing the actual results with the results they expect. However, some bugs are still not detected with manual testing. Therefore, it is desirable to automate as many of the manual testing tasks as possible by using test scripts. It is especially true when there are various kinds of computer-assisted tools for generating test scripts. But these scripts lack the common senses of a human being. Manual testers are still needed to test the high-risk areas of the products.

Test monkeys are often used in addition to manual tests and test scripts. Test monkeys are automated tools. Their testing actions are randomly or stochastically performed without a user's bias. They find bugs differently than manual tests do because they have no knowledge of how humans use the application. Test monkeys and automated test scripts can also be used adjunctly because test monkeys randomly assign test cases. These random actions can employ a large range of inputs. Because all the actions and inputs are automated without a human's attention, often all possible combinations could be performed to test the application. Microsoft reported that 10 to 20 percent of the bugs in Microsoft projects are found by test monkeys (Nyman 2000). Some of these bugs might not be found by other means. The following list includes some of the immediate benefits of using a simple and properly programmed test monkey:

- *It can be used to randomly find some nasty bugs that other means could miss.* Because a dumb test monkey doesn't need to know anything about the user interface of the application, it can be used to test different projects. Any change or addition of the GUI components will not affect the performance of the test monkey. It can also be used at any stage of the development cycle. Using this method toward the end of the development life cycle can shake out additional defects.

- *A test monkey will run and find bugs continuously until it crashes the system.* By pushing memory and other resources to the system's limits, it finds memory and resource leaks effectively. The application under test can be complex or simple. Running the test monkey continuously for days without failure increases the tester's confidence in the stability of the application.
- *The test monkey is a useful complementary tool for covering the gaps left by manual testing and automated testing.* When its random action finds a bug or causes the application to fail, it reminds the testers that there is an area that should be addressed by the test plan.
- *A test monkey can be set up to run on a system of any kind (old, slow, and unwanted).* It can be set up to run unattended by the end users. Testers can check its progress every day or so. If the monkey finds a bug, it is the least expensive and most unwanted bug in the product that could crash and hang the system.

On the other hand, test monkeys have trouble recognizing GUI components. They continuously and randomly perform actions—such as mouse movements, right or left mouse button clicks, and random text entries—in any area of the application. Eventually some actions will hit on all the GUI controls, change the application states, and continue to test until the application crashes. The results are unpredictable.

---

## Test Monkeys

Testers use the term *monkey* when referring to a fully automated testing tool. This tool doesn't know how to use any application, so it performs mouse clicks on the screen or key-strokes on the keyboard randomly. The test monkey is technically known to conduct stochastic testing, which is in the category of black-box testing. There are two types of test monkeys: dumb test monkey and smart test monkey.

*Dumb monkeys* have a low IQ level and have no idea what state the test application is in or what inputs are legal or illegal. They can't recognize a bug when they see one. The important thing is that they are eager to click the mouse buttons and tease the keyboard. These mouse and keyboard inputs are meaningless to the test monkeys, but the applications under test accept and process these random inputs seriously.

*Smart monkeys* can generate inputs with some knowledge to reflect expected usage. Some smart test monkeys get their product knowledge from a state table or model of the software under test. They traverse the state model and choose from among all the legal options in the current state to move to another state. Illegal actions may also be added to the state table on purpose. However, each input is considered a single event independent of the other inputs.

Although actions of the test monkeys are random and independent, testers have reported finding serious bugs with them. Reports can be found in articles and books listed in the bibliography at the end of this book (Arnold 1998; Nyman 2000).

---

The other difficulty is that test monkeys can't log much useful information about faults and failures. In that case, the engineers can't review a meaningful test log, so some of the bugs can't be reproduced later. One method for solving this problem is to run the test monkey with a debugger. When the monkey encounters a bug, the debugger halts the monkey and the developers can examine it. Some testers have used video cameras to monitor and record the monkey's actions. Even when a bug occurs, the monkey continues to test until it crashes the system. Later, testers can fast-forward the tape, reproduce the bugs, and learn how to crash an application.

Although test monkeys can find the most destructive bugs, it doesn't always recognize them. Monkey testing is not considered adequate for GUI test automation. What we need is some intelligent measures to complete a fully automated GUI test.

### **Intelligent Automation**

Unlike with test monkeys, with intelligent automation there should be knowledge of the functional interactions between a tool and an application. No time is wasted on useless mouse clicking and key stroking. When an action does not perform as expected, the problem can be identified and reported. An intelligently automated test tool needs an effective state table. It can be used to perform GUI testing, load testing, and stress testing. A test script is generated to perform the desired actions based on a particular application. Since the test script simulates a human operating the system, it should find a significant number of bugs. The development of an intelligent testing tool will require deep knowledge of testing theories, experience in management of software testing resources, and sophisticated programming skills.

When we develop such a tool, we should implement it with the capability to understand basic Windows elements, such as menus, command buttons, check boxes, radio buttons, and text boxes. Then it should be able to apply a set of predefined actions on each GUI component and make sure it's working properly. The tool will not only apply all possible inputs to test the application, it will also find combinations and sequences that human reviewers could never consider.

Test experts have the tendency to execute complex tests and find more bugs than are found with the available commercial tools (Marick 1994). The available tools create scripts that are able to conduct only simple tests. In such a situation, a test script executes a simple test and compares the outcome with a baseline. It then forgets the first execution and start another simple test. Users expect tools to conduct complex test sequences and find bugs.

The next sections will list some of the currently available GUI test tools and point out the areas in which they're inefficient.

### **Automatic GUI Testing Tools in the Marketplace**

This section will introduce some current testing tool vendors and the tools they offer. The list of the tools and companies is incomplete. Also, I don't mean to criticize the listed ones and

endorse the others. My purpose in listing these tools is not to criticize or endorse any but to establish a foundation on which we can base the development of a more effective testing infrastructure. In addition to my personal experience, some of the statements will be quoted from the companies' advertisements and web pages. Others are based on the observation of other test engineers. The capture/playback approach will be repeatedly referred to. The reviews are brief; only the major features of each tool mentioned will be covered.

In general, no matter what the tool vendors claim, all the test automation projects in which these tools are used are programming projects. The tools record the mouse and key events operated by human users and translate these events into computer programs instead of hand-writing code directly. Because each GUI component requires verification points and data storage, users need to invoke functionalities from different programs. For example, users need to interact with the application under test in order to record test scripts and with the tools to insert checkpoints and test cases. These interactions are manual and tedious. Let's look at a few of them here.

### **CompuWare TestPartner**

TestPartner provides functions for testing GUI and non-GUI components. Users can use it to test client- or server-side COM objects through scripts written in Microsoft Visual Basic, so testers can use it before the GUI has been created. If a tester is experienced in programming, they can write test script in VB. Testers who don't have programming knowledge can use the TestPartner Visual Navigator to create GUI test scripts and execute them. Test cases and checkpoints must be coded by the tester. If the tester does not have programming knowledge, they must ask for help in order to perform effective tests to find bugs.

For more information on CompuWare testing tools, visit [www.compuware.com](http://www.compuware.com).

### **IBM Rational Test Tools**

Rational Software has become a branch of IBM. It manufactures tools for testing applications at any stage of the software development life cycle. The tools can be used for requirements management, visual modeling, and configuration and source code control as well as testing. Rational software has acquired various vendors' products over the years, such as Visual Test, SQA, PureAtria, and Performance Awareness. For GUI testing, Rational Robot and Visual Test are often used.

#### **Rational Robot**

Rational Robot works with Microsoft Visual Studio 6. To conduct a GUI test, this tool depends on the user to activate the capture/playback recorder. The tool translates the user's action into a basic script in a special language, SQABasic. The syntax of SQABasic is similar to Visual Basic. This language supports data types including String, Integer, Long, Variant, Single, Double, Currency, and other User-Defined. It doesn't have enough functions to effectively capture and verify objects.

**Rational Visual Test**

Rational Visual Test is not a part of the other bundled Rational tools. It was first developed by Microsoft and then acquired by Rational Software. It has a user interface and allows users to develop test programs and manage test suites. Users can write test programs by hand or use the Scenario Recorder to record test sessions with the same capture/playback approach. The script language is called TestBasic. It creates test scripts for testing applications developed with Microsoft Visual Studio 6. The test program recorded by the Scenario Recorder often requires users to insert test data and verification points. It also has many hard-coded GUI coordinates. Although testers have found that this tool performs more effective testing tasks than other tools do, it is reported that this tool is not easy to work with.

For more information for Rational tools, please refer to [www.rational.com](http://www.rational.com).

**Mercury Interactive Tools**

Mercury Interactive provides products to test custom, packaged, and Internet applications. Testers often use WinRunner and LoadRunner for GUI testing.

**WinRunner**

This is Mercury's capture/playback tool, again, for Windows, Web, and applications accessed through a terminal emulation (e.g., 3270, 5250). Users can choose to record test scripts based on objects or on analogical screen coordinates. They can then specifically add or remove GUI components from an existing GUI Map, interact with wizards to check window and object bit-maps, and insert checkpoints and testing data. The scripts are recorded in a language named Test Script Language (TSL). The tool can use external database to test against the application. But the creation of data is purely manual using a wizard. Some of the features do not work for some applications or under some conditions. The recorder can not be used to create a full GUI Map to test against large applications.

**LoadRunner**

This tool uses scripts generated by other Mercury Interactive tools to perform actions a real person would do. Mercury Interactive refers to this feature as a virtual user, or vuser. Thus, LoadRunner conducts load or stress testing for both Windows and Unix environments and can be used to measure the responsiveness of the system. This tool can simulate thousands of virtual users to find locations of system bottlenecks. This allows performance issues to be addressed before release.

For more detail on Mercury Interactive tools, visit [www.mercuryinteractive.com](http://www.mercuryinteractive.com).

## Segue's SilkTest

SilkTest is a GUI testing tool from Segue. This tool runs on the Windows platform and interrogates and tests objects and GUI components created with the standard Microsoft Foundation Class (MFC) library. It uses some extensions to deal with non-MFC GUI components. To conduct a test, SilkTest provides capture/playback and a few wizards to manually interact with the application under test. When users select GUI components to test, the tool can inspect their identifiers, locations, and physical tags. As the users continue to operate the application, the tool interprets their actions into test scripts based on object properties or screen coordinates of the components. The scripts are in a script language called 4Test.

During the recording session, the users need to use the provided wizards to check bitmaps and insert verification statements. The vendor claims that this is a data-driven tool. In fact, the test code, as well as the test data, must be hand-coded. Thus, the manually created data drives only the execution of the already coded scripts.

For other tools from Segue Software, you can go to [www.segue.com](http://www.segue.com).

## Open Source GUI Test Tools

Most likely, the open source GUI testing tools are for Java developers. Many testers are developing useful tools for the open source community. Here I will list only a few of them with a brief description. For more information on open source GUI testing tools, you can visit [www.qfs.de/en/qftestJUI](http://www.qfs.de/en/qftestJUI), [www.manageability.org](http://www.manageability.org), [www.testingfaqs.org](http://www.testingfaqs.org), and [www.opensourcetesting.org](http://www.opensourcetesting.org).

### Abbot

The Abbot framework performs GUI unit testing and functional testing for Java applications. A Java test library has been implemented with methods to reproduce user actions and examine the state of GUI components through test scripts. The user is required to write the tests scripts in Java code. It also provides an interface to use a script to control the event playback in order to enhance the integration and functional testing.

### GUITAR

GUITAR is a GUI testing framework that presents a unified solution to the GUI testing problem. Emphasis of this tool has been on developing new event-based tools and techniques for various phases of GUI testing. It contains a test case generator to generate test cases. A replayer plug-in is responsible for executing the test against these test cases. A regression tester plug-in is used to efficiently perform regression testing.

### **Pounder**

Pounder is an open source tool for automating Java GUI tests. It includes separate windows for users to record test scripts and to examine the results of a test. The approaches for loading GUIs to test, generating scripts, and executing the scripts are similar to the approaches used by other tools.

### **qftestJUI**

Java programmers use qftestJUI on Windows and all major Unix systems with Java Developer's Kits (JDKs) from Sun or IBM. It is used for the creation, execution, and management of automated tests to test GUI components with Java/Swing applications. The tool is written in Java. Test scripts are also hand-coded in Java.

## **Advantages and Disadvantages of the Commercial Testing Tools**

After the preceding discussion, you can see that the available tools all have the general functions to recognize GUI components once the test scripts are recorded. Test scripts can be written by interpreting testers' actions into script languages. Later, they can be edited to include more testing and verifying functions. For testing applications, the tools can record the test scripts based on object attributes. Most of them use the combination of GUI and parent GUI components to identify the GUI of interest. The tools can read testing data from external sources and execute multiple test cases to find problems.

However, all the test projects using the available testing tools are totally manual. Only the execution is automatic if the test scripts are properly programmed. The test script is either written by hand or recorded through a capture/playback approach. Recorded test scripts must be edited and debugged for verification and checkpoint insertions. Testing data must be composed by the users. Thus, effectively using the commercial testing tools is a job for programmers, not for the non-programmers. Furthermore, the tools aren't capable of finding defects deep in the non-GUI modules.

### **Computer-Assisted GUI Testing**

I will start with the good points of the available testing tools. They assist test engineers with regard to recording test scripts, inserting verification points, and testing data, as described in the following examples:

- *When capture/playback is turned on, it can record the actions performed by a real person.* It saves testers from writing test scripts by hand, although the scripts only perform the actions and don't include the functions for verifying the tests. Later, testers can revise the recorded test script, remove the undesired actions, add verification functions, and change the coordinate-based statements to object-based statements. This way is quicker than developing a test script from scratch.

- *Testing tools can record quick test scripts at the early stages of the software development life cycle.* For example, when a GUI prototype of an application is available, the capture/playback feature can record a test script more smoothly than at the later stage when more modules with bugs are integrated. The test script can be maintained and edited later with more testing functions.
- *The testing tools also have test harnesses to manage the automated test scripts.* Testers can use the available harnesses to schedule testing executions and manage effective regression testing by reusing the script library.
- *In addition, testers can use these tools as learning assistants.* Testers can use the tool to learn the properties of different GUI components and learn how to operate different GUI components programmatically. During the process of editing the recorded test scripts, the testers will get familiar with syntax of the script language and structures of the test scripts. An experienced tester discards the capture/playback feature but uses the test script manager in a testing tool.

### **The Common Features of Capture/Playback**

The capture/playback approach records test scripts by recognizing GUI elements by their ID or by a combination of attribute values. Any change to the attributes will deteriorate the scripts from running. When this happens, the test scripts need to be rerecorded, edited and debugged, and it's possible that all the testing data created for the previous test scripts would be obsolete. Continually regenerating test scripts is time consuming.

Testing tool vendors all claim their tools include the capability of recognizing object-based GUI components. However, testers often find that a tool might recognize GUI objects created within one development environment and not recognize objects created within other environments. The reusability rate of the test scripts recorded by capture/playback is usually low. Some of the tool vendors may offer add-ins to support different development systems. Otherwise, the testers have to purchase various testing tools to meet different testing requirements of a software project.

On the other hand, the capture/playback approach assumes the GUI under test can already be functional. When the assumption doesn't hold, the tester has to abandon the current session of recording and report the problem. After the developers fix the problem, the tester can restart the capture/playback. This process of reporting and fixing may be repeated again and again. When a complete test script is finally recorded for an application, many bugs have been detected during the recording process. The effectiveness of executing the automated test script is limited, but the test script is useful for regression testing.

## Editing the Recorded Test Script

Of the bugs found using the available automated testing effort, most are found during the process of recording the test scripts. The recording process is totally manual. The testers are well aware that recorded test scripts have hard-coded coordinates to locate GUI components. These coordinate values should be removed and changed to recognizable objects. A recorded test script performs actions in exactly the same sequence they were performed during recording. Playing back the raw script will only test whether the actions happen. It doesn't have code to verify the actions performed by GUI components when users trigger them with keystrokes or a mouse.

With GUI testing, you not only need to test whether a GUI component can be manipulated by a mouse click and a keystroke, you must also test whether the desired functions are invoked correctly. The GUI test involves the execution of both GUI and non-GUI components. For example, when a Save button is tested, the test needs to check whether a file is saved and whether the assigned filename is in the correct folder. It also needs to verify whether the contents of the file are consistent. The GUI testing tools normally cannot perform these functions with raw recorded test scripts.

Thus, after each session of manual capture/playback, the tool users need to edit the recorded test script to remove the undesired actions performed during recording, change hard-coded testing values and GUI coordinates, and add functions to catch the state of the module changes behind the GUI invocation. Also, testers need to add code to enable the test scripts to predict the consequences of each GUI action and verify the invocation of the other components. Finally, testers learn from the script editing and debugging process, write test scripts by hand, and discard the capture/playback tool.

## Implementing Testability Hooks

One of the reasons the test scripts recorded by the commercial tools fail is that applications are developed with different development environments. There are various types of GUI components. For example, using Microsoft techniques, developers can choose to populate the GUI of an application with .NET, MFC, ActiveX, or third-party GUI components. Java developers can use Java or Java Swing interchangeably. Sometimes the developers assign names or IDs to components. Other times, they just accept the default values assigned by the integrated development environment (IDE). The complexity of the GUI also makes it difficult for the test scripts to work.

In defense, tool vendors claim that programmers should be disciplined and stick to the software design specifications. The tool would be able to test the applications if developers added testability hooks in the applications. The developers should insert code to call a test recorder where meaningful operations are implemented. The code would also pass all the parameters needed to re-create the interaction with the capture/playback process. Thus, the development disciplines should comply with the testing tools.

In the real world, it is impractical to implement testability hooks in applications. Testers want to operate the testing tools independently from operating the application. Developers don't want a bunch of test code mixed in with the application. They want to develop clean code. Extra code complicates the application and introduces unnecessary bugs. Besides, all software products have undiscovered bugs. Testability hooks in an application will increase the possibility of introducing the bugs in the tools into the application under test. The final result is that these testing tools are often left on the shelf.

## **Reusability for Regression Testing**

Once a program passes a test script, it is unlikely to fail that test in the future. The test scripts don't find bugs by testing against one set of testing data. They find bugs by running against different test cases. Testers should spend more time on generating creative testing data and executing the test to cover many branches of the application rather than operating the testing tools to create and debug test scripts. Effective test cases and multiple executions will increase the reusability of the test scripts.

Applications are subject to change throughout the software development life cycle. Whenever new lines of code are modified, removed, or added, the changes could adversely affect the performance of the application, so the test scripts need to be rerun. Thus, the recorded test scripts should be made useful for regression testing.

## **The Proposed GUI Testing Approach**

As you have learned, the available GUI testing tools don't meet the testing requirements to detect as many bugs as possible. Undetected bugs in software can cause economic losses and sometimes what can seem like disasters to the end users. There is a great need for reliable test technology. In this book, I'll present GUI testing methods that will actively look for components, generate testing data based on individual GUIs, drive the script generation with the testing data, and execute the test to report bugs.

### **Active GUI Test Approach**

All the capture/playback-powered testing tools depend on human users to spot GUI components. They then record the actions performed on the GUI components. Today's testing tools don't have the capability to see and apply actions to the GUI components before a test script is created. The script and data generation is all passive. Many of the tools are shipped with a test monkey. A test monkey doesn't understand software applications and performs software testing by applying random mouse and keyboard actions.

The Microsoft Visual Studio 6/.NET packages are bundled with a Microsoft Spy++ tool. This tool can spot a GUI component with a mouse movement. But it is handicapped without the capability of applying mouse or key actions. This book will introduce a method to use a hybrid of a test monkey and the Spy++. A tool with the arms of the test monkey and the eyes of Microsoft Spy++ will be able to operate the mouse actions, press the keys, and see the GUI components on the screen.

Fortunately, almost all the available testing tools have already been implemented with the capability of translating the mouse and key actions into programs. Once the arms and eyes are implanted into the testing tools, they should be able to write test scripts without the passive capture/playback procedure. Thus, an actively automated test tool can be developed.

The approach in this book to developing an active testing tool will be based on the manual testing experiences of an organization. I will begin with testing simple applications to familiarize you with manual and exploratory testing. Then I will use the knowledge gained during the manual and exploratory testing to create test data. Whenever the improved tool encounters a GUI component, it will comprehend the properties of the GUI component and foresee the consequences when a certain action is applied to this GUI component. The properties of the GUI components and the foreseeable consequences will all be stored in a data sheet. This data sheet can be used to drive the execution of the test scripts. Because the test scripts know what the applications are supposed to do, they can be executed to detect bugs when the applications are doing the wrong thing.

As the development cycle progresses in this book, we can add new features to the tool. Test script generation will eventually become unattended by human engineers and can be continued day and night. Thus, testers will be freed from recording, editing, and debugging test scripts. They can devote their time to generating test data and executing the scripts to test the application as thoroughly as possible. If some features can not be tested automatically at that time and are high-risk areas, the tester can have more time to manually test these areas. Later the tester can enable the tool with new testing requirements based on the manual testing experience.

## **Generating Testing Data First**

Each of the available testing tools has its own format for data store. Some less-effective tools can't read external test cases and their test values are hard-coded in their test scripts. The method described in this book is able to conduct an active survey, so the properties of the GUI components will be collected and saved in a popular data format, such as XML or Microsoft Excel document. Data saved in an XML document is easy to handle and review.

Once the GUI information is collected, the tool will be able to understand and predict the GUI behaviors. The tool can use GUI information to generate sequences of testing values and expected outcomes. For example, if the application requires a number as input for a certain

state, a random number will be automatically assigned into the data store. If a string is needed, the tool will guess a string of text that makes sense to the testers and the developers. If other types of data are required, this tool can initialize appropriate objects of the data type when writing the test script.

After the data generation, the tool will be able to provide testers with a chance to view the GUI test cases. Testers can choose to accept the automatically generated test cases or modify the data store immediately. They can also make multiple copies of the data store and assign different values to each test case. It is believed that once the test script executes, it will not find bugs by running against the same test cases. Multiple copies of the data store will enable the scripts to test as many branches of the application as possible, thus maximizing the possibility of finding bugs.

## Data-Driven Test Scripts

All the available testing tools have at least two features to brag about: one is their capture/playback capability, the other is their capability for data-driven script generation. However, in real testing projects, the capture/playback feature has often been reported to record less-effective scripts. Many times, the recorded scripts fail to test the application. The data-driven test script is also on the wish list of tool vendors because many testing tools don't know how to generate testing data automatically. When they record scripts, they ask the users to enter testing data via a wizard. The wizard doesn't have the power to automate the process. In other words, data is often generated after test code in the scripts using these tools, and test scripts are not data-driven.

In this book, the approach will be to conduct a GUI survey of the application under test. After the survey, the tool will collect the properties of the GUIs in a data store. It will also generate testing values in the store. It will then provide an interface for the testers to view and modify the data. After the testers confirm their satisfaction with the data, the tool will save the data in a data store that is independent of the test scripts. Thus, the test script generation and execution is a data-driven process.

GUI invocation is different than invoking a member from a non-GUI component. For example, when a GUI button is clicked, a series of subsequent non-GUI actions can be triggered. It can also cause new windows to appear. To save a file, an application may have a Save button. Clicking the Save button causes a save file dialog box to pop up. You assign a filename to a file folder and click the OK button. For a human user, the task is completed. A test script recorded by conventional tools can perform these actions. But for a manual tester, you need to confirm that the save file dialog box appears after the Save button is clicked. Then you need to verify whether a new filename appears in the folder and whether the content of the new file is as expected. All this is done by comparing the actual results with the expected

results. To accomplish the GUI actions and subsequent confirmation and verification, this book proposes an automatic test-scripting process that generates code to achieve the following testing tasks:

- The test script initiates a button click.
- It confirms the expected subsequence.
- It verifies the desired outcome.

If any unexpected event happens in one of the three steps, the result reports an error and the test script continues to test the next GUI component. When all the test cases are executed and all three steps are performed, bugs are reported to the developers.

## Summary

The requirement for quicker development and testing cycles for software projects has led to the creation of more effective GUI testing tools. This chapter discussed today's popular GUI test infrastructure briefly. Many of the GUI testing tasks are still accomplished by manual tests. Test engineers believe that a test monkey is an effective and automated testing tool, but the random actions of test monkeys cannot effectively find many of the bugs. There remains a lot of room for improvement with the currently available GUI testing tools. For example, they lack the automatic generation of testing data and test scripts.

This chapter included brief descriptions of some of the GUI testing tools. Most of these tools rely on the capture/playback process to record test scripts. Other tools require testers to write and debug test scripts by hand. These methods of script generation are not effective. The generated scripts are not reliable and become obsolete easily, and it's expensive to maintain them.

I also introduced an active method for GUI survey of an application to improve the current GUI test infrastructure. I then proposed to use the survey result to generate testing data automatically and use the generated data to drive the script execution and to achieve a fully automated testing. The ultimate goal is to dramatically shorten the time required to achieve a higher-quality software application.

In my previous book, *Effective Software Test Automation: Developing an Automated Software Testing Tool* (Sybex 2004), I introduced a tool that achieved testing of non-GUI components of an application with full automation. In that book, *full automation* means that users feed the testing tool with an application and the automated testing tool delivers a bug report. Continuing with concepts in the previous book, in the upcoming chapters of this book I will discuss methods and approaches to building a tool and accomplishing GUI testing with full automation.

The methods of testing GUI components are different than the methods for testing non-GUI components. The techniques involve a discussion of the Win32 API and advanced .NET programming. In Chapter 3 we will develop a C# API Text Viewer to help in the development of a GUI test library throughout the book. The C# API Text Viewer is a GUI-rich application and will also be used under various development stages as the application to be tested manually and automatically by the GUI testing tool. Chapter 4 will start laying out the foundation of the GUI test library. In it, you will be introduced to some useful C#. NET technologies, focusing on GUI testing automation. These GUI testing features will be demonstrated on testing the C# API Text Viewer. You'll notice that some of the programming techniques introduced in Chapter 5 have already been used in Chapters 3 and 4. If you have written programs in C#, you should not have problems understanding Chapters 3 and 4. Thereafter, in Chapters 6, 7, and 8 we will enable the tool to conduct the first GUI testing with minimum human interaction. The rest of the book will address some specific GUI testing tasks and present methods to expand this tool for more testing capabilities.