An Automated Testing System for Telephony Software A Case Study

by

Yingxiang (Ingrid) Zhou

B.Sc., University of Victoria, 2003

A Thesis Submitted in Partial Fulfillment
of the Requirements for the Degree of

MASTER OF SCIENCE

In the Department of Computer Science

An Automated Testing System for Telephony Software A Case Study

by

Yingxiang (Ingrid) Zhou

B.Sc., University of Victoria, 2003

Supervisory Committee

Dr. Hausi A. Müller, Supervisor (Department of Computer Science)

Dr. Dale Olesky, Departmental Member (Department of Computer Science)

Dr. Frank D. K. Roberts, Departmental Member (Department of Computer Science)

Supervisory Committee

Dr. Hausi A. Müller, Supervisor (Department of Computer Science)

Dr. Dale Olesky, Departmental Member (Department of Computer Science)

Dr. Frank D. K. Roberts, Departmental Member (Department of Computer Science)

Abstract

As the complexity of software system increases, delivering quality software successfully becomes an ever more challenging task. Applying automated testing techniques effectively to the software development process can reduce software testing effort substantially and assure software quality cost-effectively. Thus the future of software testing will rely heavily on automated testing techniques.

This thesis describes a practical approach to automated software testing by investigating and analyzing different automation test tools in real-world situations. In view of the fact that the key to successful automated testing is planning, understanding the requirements for automated testing and effectively planning is critical and essential.

This thesis presents the design and implementation of an automated testing framework. It consists of an automated testing tool, which is based on the commercial product TestComplete, as well as associated testing processes. The application area is telephony communications software. To demonstrate the viability of our automated testing approach, we apply our testing framework to a Voice-over-IP telephony application called Desktop Assistant. This case study illustrates the benefits and limitations of our automated testing approach effectively.

Table of Contents

List of Tables

List of Figures

Acknowledgments

Special thanks to my supervisor, Dr. Hausi A. Müller, for his patience, guidance, support, and inspiration throughout this research. I appreciate and cherish the great opportunity to work in his research group and pursue my graduate studies under his supervision. I would also like to thank my committee members, Dr. Frank D. K. Roberts, Dr. Dale Olesky, and Dr. Kin Fun Li, for their valuable time and effort.

I am also grateful to all the members of the Rigi research group for their contributions. In particular, I would like to acknowledge the help I received from Qin Zhu, Grace Gui, Holger Kienle, Piotr Kaminski, Scott Brousseau, Jing Zhou, Tony Lin, and Feng Zou.

I also would acknowledge the generous support of NewHeights Software Corporation and my work supervisor, Craig Hansen, for granting me the chance to develop the automated testing system.

Finally, I would like to thank my friends and family for helping me through this long process with their love and care.

Dedication

To my parents

Chapter 1
Introduction

This thesis reports on the development of our automated testing framework for telephony communications software. In particular, we designed, implemented, tested, and deployed an automated testing system and processes to test a commercial product of NewHeights Software Corporation called Desktop Assistant. This product is a Voice-over-IP (VoIP) telephony software application that facilitates office communication. Through its interface a user can manage contacts, instant messaging, and call status. To develop our automated testing system, we investigated and analyzed various automated testing tools and methodologies. The automation tool finally chosen as the core component of our automated testing system was TestComplete by the company AutomatedQA [1].

1.1 Motivation

As flexible as our relationship with computers has become during the past half-century, at least one constant remains: Wherever you find a computing system, you will find problems and difficulties, commonly known as “bugs” [30]. The most common approach for corporations to exhibit the presence of bugs is through extensive testing [28]. Over the years, companies have developed a variety of testing methodologies and tools for many different application areas. Testing is a time-consuming and tedious process. As a result, researchers and practitioners have developed many approaches to automate testing. Automated testing has proven to be one of the most powerful strategies to improve testing efficiency. Some organizations save as much as 80% of the time it would take to test software manually [11].

Test engineers, who develop automated testing systems, usually take a software engineering approach to testing software as well as designing and developing practical automated software testing systems [28]. Such systems involve methods and tools aimed at increasing longevity, reducing maintenance, and eliminating redundancy of the software to be tested [18]. Nowadays, software testers are under considerable pressure to test an increasing amount of code more efficiently and more effectively. Test automation is a way to alleviate this pressure and be better prepared for the onslaught of old and new code to be tested. When a new version of a software system is released, thorough validation is required for newly added features

and for any modifications made to the previous version. Usually, considerable repetitious and tedious activities are required to validate functional and non-functional requirements of a subject system in a limited amount of time.

Automated testing seems ideal for this process, as, by design, computing systems perform large numbers of repeated actions. Automated testing is the use of software to control the execution of tests, the comparison of actual outcomes to predicted outcomes, the setting up of test preconditions, and other test control and test reporting functions [32]. Ideally, automated testing lets computers detect and report their own errors and problems automatically without human interaction. Therefore, when set up properly, computing systems can perform repeated activities and report testing results automatically, thereby significantly reducing manual testing efforts.

In addition to reporting on errors and problems, automated testers can deliver on other objectives. For example, when automated testing tools generate programs to execute simple and tedious testing routines, other testing resources are freed up for more challenging and complicated testing. In addition, problems arise when software functions that previously worked as specified in the requirements specification stop working or no longer work in the way planned. Known as regression problems, they occur as unintended consequences of program changes. To detect these problems, a type of test called a regression test is required. Automated testing increases the frequency of regression tests. Normally, full regression testing is a significant investment of quality

assurance resources and, consequently, occurs rarely. If the majority of regression tests are automated, then they can be performed more frequently.

To improve testing efficiency and reduce testing effort for selected telephony communications software produced by NewHeights Software Corporation, we designed and implemented an automated testing framework. As a subject system we employed Desktop Assistant, a VoIP telephony software application that facilitates office communication. Through its interface a user can manage contacts, instant messaging, and call status. We investigated and analyzed various automated testing methodologies and tools and ultimately to use a tool called TestComplete as the core component of our automated testing framework. Using TestComplete, we designed and implemented an automated testing system that reduces manual testing significantly.

1.2 Approach

To test a piece of software, one needs to analyze the software and investigate its functions through test cases. A test case requires the investigation of a specified set of actions performed by a particular function of the software and of the outputs following those actions. If the output is as expected, the test case awards a pass. If not, it generates a fail. If one is planning to perform automated testing, one also needs to examine the test cases to which automated testing can apply. Since not all test cases permit automation, the tester needs to select test cases carefully. For example,

the test case for testing the clarity of audio passing through the telephony application can never be automated.

Desktop Assistant features manage phone calls, instant messaging, and contact management, allowing test through its user interface. By monitoring the user interface, one can validate its functions. For example, on receipt of a phone call, an incoming call popup window appears. Caller identification and name will show up in the window. By validating the existence of this window and the caller identification and name, one can pass or fail the test case of receiving an incoming call. Therefore, the majority of testing will, in fact, be graphical user interface (GUI) testing [25].

For automated testing, it is also important to select an appropriate tool to carry out the tests. The automation tool that we selected is called TestComplete. TestComplete is a full-featured environment for automated testing for Windows, .NET, Java, and web applications. In particular, it is capable of testing graphical user interfaces. It is a record and playback type of automated test tool [33]. It generates test pass and fail reports based on mouse actions that a user performs on computing systems with the subject software application. The results allow editing and maintenance of the programs.

We chose TestComplete as our automated tool for several reasons [1]. First, the software under test is a .NET application. Thus, TestComplete is capable of testing it. Second, the

majority of testing for Desktop Assistant is GUI testing and TestComplete is an excellent tool for this type of testing. Third, TestComplete provides a user-friendly interface for the management of test suites, for editing of programs tested, for maintenance of programs, and for producing records of tests and their results. Finally, TestComplete is reliable; it captures images in pixels and compares them precisely with expected images.

1.3 Thesis Outline

Chapter 2 of this thesis describes selected background for software testing, including manual testing and automated testing as well as existing automated testing methods and tools. Chapter 3 elicits requirements for an automated testing framework including test case requirements and precondition setup. Chapters 4 and 5 describe the design and implementation of our automated testing system. Chapter 6 evaluates our automated testing system. Chapter 7 summarizes the thesis, highlights the contributions of the work reported in this thesis, and finally suggests selected further work on this topic.

Chapter 2
Background

This chapter describes background for software testing, including manual and automated testing, and existing automated testing methods and tools. We begin with a description of the notions of software life cycle, software quality, software bugs, and test automation. We then describe different practical automation testing methods and tools, with a focus on keyword-driven or table-driven automation frameworks, data-driven frameworks, and record and playback types of test automation [33].

Terminology

2.1.1 Software Life Cycle

A software life cycle is the sequence of phases in which a project specifies, designs, implements, tests, and maintains a piece of software [23]. One can use different software life cycle models to describe the relationship between these phases. Common software life cycle models include the waterfall model, the extreme programming model, the throwaway prototyping model, the spiral model, the evolutionary prototyping model, and the open source software development model [30].

We use the traditional waterfall model [4, 28], as depicted in Figure 2-1, to show how software testing occurs in a software life cycle. We have found this model to be useful in practice because it makes explicit the testing inherent in every phase of software development. If there are risks involved in the development process (e.g., chance of unknown requirements or requirements being added later on), then the spiral model (i.e., several iterations of the waterfall model where each iteration builds a little, evaluates a little, and tests a little) is preferred over the waterfall model. However, if there are no perceived risks, then the waterfall model is adequate and works well.

In the waterfall model, software development follows a standard set of stages. The first phase is typically referred to as requirements engineering. The design phases involve architecture design, namely, breaking the system into pieces and performing a detailed study in which each piece is assigned a place in the overall architecture. An implementation phase follows the design phase, requiring the writing of programs for system components and testing them individually. The implementation phase involves further program writing, removal of problems, and unit testing. After implementation, the pieces require integration and the whole system demands testing, known as integration, system, or acceptance testing. Finally, the system is ready for deployment and operation.

In a development process using the waterfall model, each phase is separate and the phases follow each other. While there are feedback loops between any two phases, developers typically do not advance to the next phase until the previous phase is reasonably complete. For instance, the developer completes the requirements specification before starting the design; one cannot start writing programs until the design of the system is completed. As a result, it is important to have a number of solid design reviews. These reviews typically include a requirements review, a conceptual design review, a preliminary design review, and a critical design review.

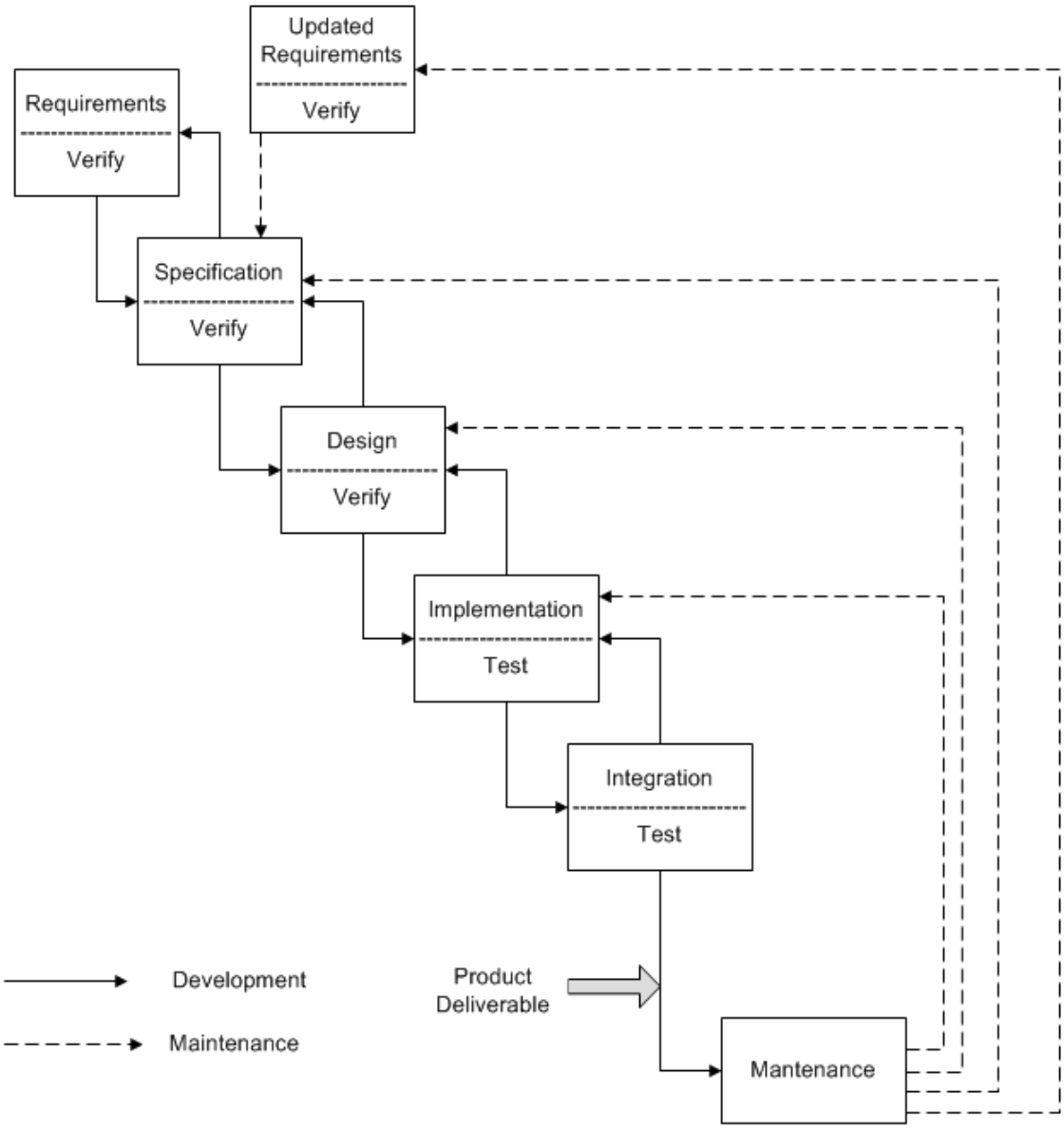

Figure 2-1: Waterfall Model

Opportunities for automated testing arise during the implementation, integration, and maintenance phases of the waterfall model [28]. For a large and complex system, the developer usually uses tools or writes a program to assemble all the system files and makes another

program that will install these files to a customer's workstation. This process is called a build-process, which is usually automated. The developer can integrate unit testing into the build-process and, likewise, automate unit testing. Each time the build process runs, unit testing also occurs. This can help reduce the testing effort, as one can discover problems as one builds the system. During the testing phase, the developer may use different automation methods as discussed in Section 2.2 below.

2.1.2 Software Quality

Software quality measures how well a piece of software meets the pre-defined requirement specifications. Good quality signifies that software conforms to the specified design requirements. Typically, one measures the degree to which a piece of software satisfies criteria such as usability, reusability, extensibility, compatibility, functionality, correctness, robustness, efficiency, and timeliness.

Usability refers to the ease with which one can use the software. If software is highly useable, people with different backgrounds and with various qualifications can learn to use it easily. Reusability is another software quality factor. It refers to the ability of software elements to serve in different applications. In addition, when software requirements change, software needs updating or requires extensions. The ability of software to change to match changes in

requirements is called extensibility. Compatibility defines how well software elements combine with each other. For instance, a piece of software is usually required to be compatible with different operating systems. Compatibility also includes the demand for a piece of software to work with various hardware components. In information technology, functionality is the total of what a product, such as a software application or hardware device, can do for a user [35]. Functionality measures the ability of a piece of software to solve problems, perform tasks, or improve performance.

In addition to these software quality criteria, software must be correct, robust, efficient, and timely. Correctness measures how well software products perform their specified tasks. In other words, correctness defines whether the software performs the expected behaviour. As well, a software system must usually handle abnormal situations. Robustness measures how software systems act under abnormal circumstances. The amount of resources needed to run a software system, such as computing time, memory usage, and bandwidth determines efficiency. Finally, the time of release of a system to a customer is important; the timeliness of a product, that is, its release date, is usually part of the software requirement specification.

2.1.3 Software Defects

A software bug or defect is “an error, flaw, mistake, undocumented feature, failure, or fault in a computer system that prevents it from behaving as intended” [30]. Software defects can originate in the requirements specification, the design, or the implementation of a software system. The causes of defects are human errors or mistakes made during requirements specification, design, and program writing.

Software bugs usually affect users of programs negatively and the effects may appear at different levels. Some defects may produce such minor effects on some function of the system that they will not come to the user’s attention. However, some software bugs can cause a plane to crash or cause a bank to lose large amounts of money. Some of the worst software bugs in history are well-documented [16, 30].

July 28, 1962—Mariner I space probe. A bug in the flight software for Mariner I caused the rocket to divert from its intended path on launch. Mission Control destroyed the rocket over the Atlantic Ocean. The investigation into the accident uncovered that a formula written on paper in pencil was improperly transcribed into computer code, causing the computer to miscalculate the rocket’s trajectory.

1982—Soviet gas pipeline. Operators working for the Central Intelligence Agency (CIA) allegedly planted a bug in a Canadian computer system purchased to control the trans-Siberian gas pipeline. The Soviets had obtained the system as part of an effort to purchase covertly or steal sensitive US technology. The CIA reportedly found out about the program and decided to make it fail through equipment that would pass Soviet inspection and fail in operation. The resulting event was reportedly the largest non-nuclear explosion in the planet's history.

Bugs are a consequence of human errors in the specification, design, and programming task and are inevitable in software development [30]. However, developers can reduce the frequency of bugs in programs by adhering to systematic programming techniques, software engineering methods, processes and tools, as well as programming language, development environment and operating system support. Another way to reduce bugs is to test software thoroughly using the best testing methods and techniques [28]. Consequently, efforts to reduce software defects often focus on improving the testing of software [12]. However, testing of software only proves the presence of bugs and no amount of testing will ever prove the complete absence of bugs [7].

2.1.4 Test Automation

The idea of test automation is to let computing systems detect their own problems. More specifically, test automation is the use of software to control the execution of tests, the comparison of actual outcomes to predicted outcomes, the setting up of test preconditions, and other test control and test reporting functions [18]. The goal of test automation is to improve the efficiency of testing and reduce testing effort and cost. Since test automation requires automation software, differentiation of actual outcomes from expected outcomes, preconditions, and test result reports, the automation process itself becomes a software development process involving a software life cycle. As with the development of any software, most successful automated testing developers use a systematic software engineering approach to develop systems to automate the testing process [18].

However, test automation may fail for a number of reasons: (1) spare time automation, which means that testers do automation in their spare time without regular working time allocated specifically for automation; (2) lack of clear development goals; and (3) lack of developer experience. Resources are seldom allocated to test automation, as testers usually have a heavy workload. As a result, test automation is typically a project undertaken in the spare time of a developer and therefore receives inadequate time and focus. In any case, test automation has many different motivations. It can save time, make testing easier, and improve testing coverage.

Yet, these diverse goals can lead automation in different directions, and, in the end, it may fail because the goals are unclear. Accordingly, it is important to identify goals and specifications for testing automation clearly. In addition, programmers with little experience tend to create test automation projects that are hard to maintain and thus will likely lead to failure. Finally, software development firms have significant personnel turnover, reducing the time a developer may have to learn the needs of specific products requiring testing. At times, the primary goal of meeting testing requirements may be disregarded, which eventually may cause a test automation project to fail [26].

Since test automation seems most successful when automated testers take a software engineering approach to the development of automation software, it is important that each phase of the software life cycle in the test automation project receives sufficient attention. Testing requirement specifications need proper documentation; good design is essential; identification of test criteria is required. We discuss how to improve test automation in Section 2.3 below.

Since test automation seems successful when automated testers take the approach of software development to accomplish it, it is important that each phase of the software life cycle in the test automation project gets enough attention. Requirements specifications for testing need to be well documented. Good design is also very important for the success of test automation.

2.2 Automated Testing Methods and Tools

The automated testing tools include keyword-driven or table-driven testing frameworks, data-driven frameworks, record and playback tools, code analyzers, coverage analyzers, memory analyzers, load and performance test tools, and web test tools [33]. However, for this case study, only keyword-driven or table-driven automation frameworks, data-driven frameworks, and record and playback types of test automation were deemed suitable.

2.2.1 Keyword-Driven or Table-Driven Testing Automation Framework

Keyword-driven testing and table-driven testing are interchangeable terms that refer to an application-independent automation framework [33]. In a keyword-driven test, the functions of the tested software are written into a table with systematic instructions for each test. Then test programs read the instructions from this table, execute them, and produce outcomes. The outcomes produced are compared with pre-written expected behaviour and test reports are written based on the compared results. Thus, in a table-driven automation framework, a test driver reads a table, executes the instructions from the table, compares the actual outcomes with the expected outcomes, and writes the test results.

Keyword-driven testing is application-independent, which means that the development of data tables and keywords is completely independent of test drivers. All steps for performing a function are written in a table before feeding into a test driver. Thus, the test driver does not include any of the steps to perform the functions. Test drivers include the automated tools used to execute these data tables and keywords as well as the test script code that “drives” the tested application and the data. To execute or “drive” the data, test drivers contain methods that read a table of instructions as well as methods that run these instructions after read-in on the tested software. After the data-tables and keywords have been developed, test drivers execute the application under test, in conjunction with the input written into the data-tables.

Table 2.1 is a sample data-table created for the testing of a calculator program [24]. In this table, the Window column corresponds to the calculator form where the action is performed. The Control column represents the control where the mouse is clicked. The Action column indicates the action taken with the mouse. The Arguments column shows the name of a specific control on the calculator such as 1, 2, 3, 4, or +. The test driver then reads each step, executes the step on the tested application based on the keywords. It also performs error checking, and records any relevant information.

Table 2-1: Calculator Data Table for Keyword-Driven or Table-Driven Testing

Window	Control	Action	Arguments
Calculator	Menu		View, Standard
Calculator	Pushbutton	Click	1

Calculator	Pushbutton	Click	+
Calculator	Pushbutton	Click	3
Calculator	Pushbutton	Click	=
Calculator		Verify Result	4
Calculator		Clear	
Calculator	Pushbutton	Click	6
Calculator	Pushbutton	Click	-
Calculator	Pushbutton	Click	3
Calculator	Pushbutton	Click	=
Calculator		Verify Result	3

Once a data table has been created, a simple program can be created to perform tests. Figure 2-2 presents the script for a program used as the test driver for the test data listed in Table 2-1 [24].

```
Main script / program
Connect to data tables.
Read in row and parse out values.
Pass values to appropriate functions.
Close connection to data tables.
Menu module
Set focus to window.
Select the menu pad option.
Return.
Pushbutton Module
Set focus to window.
Push the button based on argument.
Return.
Verify Result Module.
Set focus to window.
Get contents from label
Compare contents with argument value.
Log results
Return.
```

Figure 2-2: Pseudo-Code for Sample Driver for Testing a Calculator

Thus, Figure 2-2 demonstrates how to write a script to perform tests and in part automate testing. While manually running the test cases, the data table can be generated and coding can be done.

2.2.2 Data-Driven Testing Automation Framework

The concept of a data-driven automation framework involves using a test driver to produce input that will generate outcomes, and then compare the actual outcomes with the expected outcomes. The test driver can be an automated tool or it can be a customized testing tool. Combining this with the idea of a test case, this input is basically equivalent to the action or steps in a test case and the outcome of the expected results in a test case.

Data-driven testing depends on testing an application's effectiveness with a range of inputs or data [29]. Data-driven testing is effective when the input is huge. When a large number of combinations of input data require testing, it becomes impossible for testers to enter all of the data manually. It is then helpful for a data-driven method to generate inputs to produce and record outcomes. Note that a data-driven testing framework can test extreme conditions and invalid inputs just as a manual tester would. Moreover, a data-driven test can verify that an application responds appropriately when a number is entered that is outside of the expected range, or a string is entered in a date field, or a required field is left blank. Data driven tests are often part of model-based tests, which build up randomized tests using a wide set of input data [29]. Model-based testing is software testing in which test cases are delivered in whole or in part from a model that describes some aspects of the system under test [28].

Data-driven testing frameworks take input and output values from data files such as datapools, CVS files, or Excel files. These files are test datasets that supply realistic data values to the variables in a test program during testing. Programs include variables that hold these input values and expected outcomes. The actual outcomes can be produced while navigating the tested software. The test scripts navigate the tested software, read the data files, and record the test results. The difference between table-driven and data-driven frameworks is that in a table-driven framework, the navigation of the tested software is included in a data table; however, for a data-driven framework, it is written in a test program and only data files contain test data.

After analyzing the requirements for our testing strategy, we concluded that for our case study, Desktop Assistant, a data-driven approach is inappropriate. Testing the application does not require large amounts of data input. Boundary conditions or invalid inputs are not our main testing focus. Regression tests, however, dominate the testing effort. In the following section, we discuss in detail why we chose a record and playback approach instead of a data-driven approach as a test automation strategy for our case study.

2.2.3 Record/Playback Testing Automation Tool

A record and playback automation tool is a tool that records the actions that a user performs while interacting with the tested application; it generates test scripts and plays back the

scripts. The actions include mouse clicking and keyboard typing. The user can edit and maintain the scripts according to the testing requirements. For instance, the user can organize test cases to run sequentially and when playing results back, outcomes can be captured. The captured output can then be compared with the predefined expected outcomes. Finally, test results can be logged.

A record and playback automation tool is especially useful for regression testing when a graphical user interface (GUI) is involved. Since a user needs to interact with the interface of the application, the interface must be ready prior to the implementation of an automation test. It is impossible to record scripts against an interface that does not exist. In the workflow prior to the use of the record and playback tool, testers analyze test requirements and document them. This part is called the test plan design and writing phase. Then testers execute the test plans and report defects or bugs, which are subsequently fixed by the developers; then the testers re-run these tests to validate the implemented fixes. Afterwards, test automation begins. Automated testers can automate regression tests using the test documents as specifications. By the time developers have automated the tests, the regression tests can be executed repeatedly with ease after every change to the software.

It is relatively difficult to maintain test scripts while using record and playback tools. Therefore, it is important that the automated tester organize and manage the test suites carefully. The automation tool that we chose for our testing is *TestComplete*. We discuss and list the

reasons for choosing TestComplete as our core automation tool in Chapter 4 when introducing the tool in more detail.

2.2.4 Comparison of Practical Test Automation Tools

This section introduces different automation tools that are popular in the market. We assess these tools based on the following criteria: subject application type, usability, maintainability, supported platforms, and cost. We selected TestComplete based on this assessment.

Watir is a typical keyword-driven automation tool that targets web browser testing. It is open source and supports Internet Explorer on Windows, Firefox on Windows, Mac and Linux, and Safari on Mac platforms. It is easy to use and maintain. Obviously, it does not suit our needs as our subject application, Desktop Assistant, is not a web browser application, but rather a .NET application.

SDT's Unified TestPro is another keyword-driven automation tool for Client/Server, GUI/Web, E-Commerce, API/Embedded, mainframe, and telecom applications. This test automation tool is role-driven. Roles include *administrators* who configure projects, configure test management, allocate users, and back up projects; *designers* who create partitions, test cases and keywords, design test cases, and create data tables; *automation engineers* who capture GUI maps and implement keywords; and finally *test executors* who create and execute test sets, view

test results, and view reports. This tool is not suitable for testing our application either. The main reason is its incompatibility with .NET applications. Moreover, it is host-based, which is impractical for us to deploy the tool.

In addition to the above keyword-driven tools, there is a data-driven automation tool called TestSmith, which is used to test web sites and web applications that feature embedded Applets, Active-X Controls, and animated bitmaps. It can also be used to test Java and C++ applications.

e-Tester is a component of the e-Test Suite and uses HTML, Java, or Activate-X technologies for functional testing of web applications. E-Tester is a record/playback tool.

We discuss TestComplete specifically in Chapter 4. Table 2-2 below summarizes the discussed criteria for test automation tools. We use those criteria to evaluate these different automation tools and select TestComplete as our tool for automation test.

Table 2-2: Overview and Comparison of Test Automation Tools

Tool	Automation Framework Type	Features, subject application types	Usability	Maintainability	Platform	Cost
Watir	Keyword-Driven	Web Browser including Internet Explorer, Firefox, Safari	Easy to use	Easy to maintain	Windows 2000, XP, 2003 Server and Vista	Open Source
Unified TestPro	Keyword-Driven	multi-tier Client/Server, GUI/Web, e-Commerce testing, API/Embedded, Main-frame and Telecom/Voice testing	Roles-Based, complicate to use	hosted application; hard to maintain	Windows, Unix, Embedded systems and Telecom	$6000
TestSmith	Data-Driven	HTML/DOM, Applets, Flash, Active-X controls, animated bitmaps and Java, C++ Applications	Hard to use	Hard to maintain	Windows NT/2000/XP	Low Cost
TestComplete	Record/Playback	Delphi, Win32, .NET, Java, and Web applications.	User-friendly	Allow re-useable components to ease the maintenance	Any Platforms	$1000
e-Tester	Record/Playback	web applications that use HTML, Java or Active-X technologies	Easy to use	Hard to maintain	Windows NT, 95 and 98, XP, Vista	Low cost

2.3 Summary

This chapter presented background on software life cycle models, software testing, and characteristics of several automated methods and tools. It also described common testing automation types, such as data-driven and record and playback automation frameworks. The following chapter discusses our application-specific requirements for testing automation processes and tools.

Chapter 3
Testing Automation Requirements

Developing an automated testing system involves three main components: a subject system (i.e., the system to be tested), automated testing tools, and automation test engineers (i.e., developers, who develop the subject system; test engineers, who design and implement the testing system and associated testing processes; and testers, who test the subject system).

This chapter discusses requirements with respect to all three components of an automated testing environment in detail. Section 3.1 outlines the necessary qualifications for automation test engineers. Section 3.2 discusses non-functional requirements, such as maintainability and reliability for an automated testing environment for a particular application type—namely

telephony applications. Chapter 4 introduces *TestComplete*, an automated testing tool chosen for developing our automated testing system.

3.1 Automation Test Engineer Qualifications

This section discusses the requirements of the people involved in building automated testing systems which are then used to test specific subject software applications.

In an ideal world, the goal of automated testing is to let computing systems manage themselves and detect their own bugs. In this case, no human interaction would be necessary. However, in reality, humans play a very important role in demonstrating the presence of software defects, in developing strategies for finding bugs, and in automating the process of testing software systems for defects continually over long periods of time.

The life cycle models and software engineering processes employed for regular software development also apply for the development of an automated testing system. Therefore, assuming that we want to follow a systematic software engineering approach to develop automated testing systems, it is crucial that the test engineers understand the software development processes and have related development experience before they embark on developing an automated testing system. However, not all test engineers, who have some

development background, have the skills to design and write automation programs effectively [1, 28]. Thus, automated test engineers, who employ automation technology, must not only understand the concepts and issues involved in software testing, but also software engineering issues, such as software quality criteria, to be able to develop and maintain test automation methods, tools, and processes effectively. Moreover, test engineers must possess significant knowledge of the application domain.

Automation test engineers must have a solid understanding of the software development process. He or she must design a system with the testing requirements in mind and must also have good programming skills. Moreover, the ability to realize design goals in the implementation with long-term maintainability in mind is crucial. It is also important for an automation test engineer to understand his or her responsibility as a tester. One must have considerable knowledge about testing requirements and environments.

One should also keep in mind that the goal of automated testing is to ensure that the functions of an application under test are performing correctly and that the non-functional requirements are satisfied. Furthermore, sometimes a tester is consumed with his or her programming role and spends most of the available time on improving the system itself, while forgetting the main objective—to detect faults in the system and report them appropriately [8]. Therefore, it is essential to the success of automated testing that test engineers understand and appreciate the testing goals and take the notion of testing responsibility seriously.

3.2 Subject System Testing Requirements

Having highly qualified automation test engineers is a good starting point for successful test automation. Another important factor towards test automation success is to analyze the testing requirements of the subject software system. These requirements include the analysis of the necessary test coverage for the application under test, the maintainability and reliability of the subject system and the automated testing system, and the challenges that will be encountered during the implementation and long-term use of the automation system. Gauging and predicting the future evolution of the subject system and its testing environment to a certain extent is also critical for the design of testing tools.

3.2.1 Automation Coverage Analysis

Due to the limitations of current testing automation technology, not all tests can be automated. Furthermore, it is of little value to allocate test resources for automated testing if manual testing costs less. Therefore, it is important to evaluate the tradeoffs between automated and manual testing carefully and, in particular, determine the test cases which lend themselves to automation [1]. How to select test cases for automation essentially depends on the objectives of

test automation, the capabilities of the test automation tool at hand, and attributes of the subject system.

For most organizations employing test automation, the general objective is to reduce the testing effort. The most time-consuming tests are the regression tests, since regression testing means re-testing after corrections or modifications of the subject software or its environment. Therefore, when selecting test cases, one should focus on those test cases that are run during regular regression testing. If blocking defects are disclosed, developers need to be notified expediently. Another type of testing is acceptance testing, which occurs before tested software is deployed [8, 28, 30]. Acceptance tests need be performed quickly, frequently, and repeatedly when the product is ready for delivery. In addition, acceptance testers need to demonstrate to stakeholders that the major functional requirements of the product are satisfied as expected.

After analyzing automation coverage for a subject system, the maintainability and reliability of the automated testing system must be investigated and considered.

3.2.2 Maintainability and Reliability of Automated Testing System

Maintainability and reliability are vital for all types of software systems. Software maintainability is defined as the ease with which a software system or component can be

modified to correct faults, to improve performance, or to adapt to a changed environment [16]. When maintainability considerations are not taken into account, automated testing practices may fail [21, 22]. For instance, when low-level interfaces of an application change, the corresponding automated testing system may become difficult to maintain in that test engineers have to spend an inordinate amount of time maintaining the operation of the automated testing system and, hence, development managers have no choice but to abandon such automated testing projects [20].

It is rather challenging for test engineers to keep an automated testing system operational when the interfaces of a tested product change. Carefully analyzing and selecting an automated tool which can accommodate such changes is one solution. This problem is also known as the Vendor-Lock-In Antipattern [5, 6]. A common solution to this problem, which improves maintainability in turn, is to incorporate a level of indirection into the subject system to control the ripple effects that potentially propagate through the entire subject system as well as the test automation environment.

3.2.3 System Reliability

According to the IEEE Standard Computer Dictionary, software reliability is defined as follows [10]. Reliability is the probability that software will not cause the failure of a system for a specified time under specified conditions. The probability is a function of the inputs to and use

of the system in the software. The inputs to the system determine whether existing faults, if any, are encountered. Thus, reliability is the ability of a program to perform its required functions accurately and reproducibly under stated conditions for a specified period of time [10].

3.2.4 Challenges

Test engineers face two key challenges when implementing an automated testing environment for a subject software system: (1) *network and server stability*; and (2) *application hang or crash*. *Application hang* indicates that the application process is not responding. When the automated testing system tries to perform further actions on it, no results will be returned. *Application crash* means that an application process shuts itself down and does not allow further testing. Naturally, these two issues can cause automation programs to abort.

For example, when a network is abnormally slow or when a telephone server is unstable, an automation system can run into a state where it takes an unusual amount of time for an application window to appear. In this case, if an error handling method is not inserted into the automation system, the system will try to find the expected window. If it cannot detect the expected window within a predefined time, the execution of the programs would normally stop. Therefore, it is important to incorporate error handling methods into the testing environment to deal with such situations.

For example, we can use the reception of a call to demonstrate an error handling method. When a user receives a call, the incoming call popup window is supposed to appear within a reasonable amount of time. In the program, one specifies the system delay time to allow the programs to find the window. However, when the network is abnormally slow, it takes a different amount of time for the window to show up. When an automation program attempts to detect the window and tries to perform other operations on the window, but cannot find it, it will stop running. In this case, one must insert an error-handling method into the program to recover from the error state to be able to continue automated testing with the next test case. Figure 3-1 depicts a sample pseudo-code fragment to handle abnormal testing situations.

```
function AnswerCall
    wait for the incoming call popup window to show up for 1 second
    while (time < 1 minute and window is not showing up)
        system delay 1 second
        wait for the window to come up in 1 second
        increase the variable time by 1 second
    get the window
    perform answer call action on the popup window
```

Figure 3-1: Error Handling Test Script

In a normal situation, an incoming call window appears in, say, less than five seconds after an incoming call occurs. However, if the system experiences server stability issues, it can take much longer to get the window onto the screen. To avoid wasting time, a loop can set the upper bound to, for example, 60 seconds. As soon as the window appears, the next step will be executed.

There is no perfect method to deal with application freezing or program crashes automatically. A user has to shut down the process and restart the execution manually. The biggest concern at the moment is that testing programs tend to crash due to errors in application record files, which causes significant delays for automated testing and leads to inaccurate or uncertain results.

3.3 Summary

In conclusion, there are three main components involved in our automated testing process: automated tester, the tested software, and the automated testing tool. These three components need to perform well and interact properly with each other for automated testing to succeed. Failure in any of them will result in a breakdown in the automated testing process.

Automated test engineers must understand the concepts of automated testing well to be able to perform automated testing properly and judge the results of automated testing accurately. Testers need to understand clearly that automated testing cannot cover all testing. Automated testing improves testing efficiency, but does not replace manual testing. The automated coverage depends on the objectives of the testing and the capability of the automation tool.

Chapter 4
Automation Testing Tool TestComplete

Automated coverage depends not only on the testing requirements of the subject system, but also on the capabilities of the automation tool. This section introduces *TestComplete*, a commercial testing tool for automating regression testing, and presents its capabilities as a base tool for our automated testing environment [1].

First, we outline the main features of TestComplete and then introduce the debuggers that are instances of the tested application. The debuggers are included in the automated testing system and run on the same machine as the subject software under test. Finally, we discuss the

installation of the tested subject application including the debuggers, as well as the configuration prior to the execution of the automation test scripts.

4.1 TestComplete—A Record and Playback Automation Tool

To introduce TestComplete, we describe its features and the reasons why we selected it as our core automation tool. Typically testing plans and test cases are written to cover all the functionality specified in the requirements specification and the functional design documents. Due to the limitations of automated testing, not all test cases are suitable for automation. Test cases need careful examination, analysis, and selection with respect to automation suitability. TestComplete provides a set of basic validation techniques which can serve as a guideline for test case analysis. These techniques include image, text, button, and checkbox validation as well as existence, visibility, and time validation.

In image validation, correct images are saved to a pre-defined image directory as expected images. These images can represent windows, icons, or menus. While executing scripts against a new version of a product, run-time images are captured and saved as actual images. Expected and actual images can then be compared to validate the correctness of the actual image. This technique is precise since TestComplete captures pictures in pixels.

In text validation, all text (e.g., labels, window titles, warning messages, or instant messages) appearing in the tested subject application can be captured as actual text. A user can define the expected text in the scripts. To validate a specific text, the expected text is compared to the actual text.

In button and checkbox validation, TestComplete can inform a user if a button is enabled and if a checkbox is checked. For example, if a button is expected to be enabled in a certain situation, and TestComplete informs the user that the button is actually disabled while running, then the test case for this button will fail.

In existence or visibility validation, TestComplete can recognize whether a process, window, or an icon exists or is visible at run-time. As a result, when a test case states that the expected result is that the process, window, or icon should exist or be visible, TestComplete can verify it successfully.

Finally in time validation, TestComplete can capture timing requirements and define and implement timing constraints using loops.

In addition, compared to most available automated tools, TestComplete provides various other advantages for our project.

1. TestComplete automatically saves the project after each execution to prevent data loss and to be able to continue testing at these checkpoints later.
2. The execution routine is simple and easy to follow, which makes it easy for testers to modify programs.
3. TestComplete provides capabilities for users to manage test suites.
4. Since the tested software changes frequently, the maintenance of the automation system becomes crucial. TestComplete enables users to define libraries whose functions can be accessed and invoked everywhere in the project, so that users can encapsulate and localize modules and thereby ease maintenance of test scripts.
5. TestComplete provides a framework for users to write validation messages and report test results effectively.

Finally in time validation, TestComplete can capture timing requirements and define and implement timing constraints using loops.

With the above techniques, TestComplete can be used to test much of the functionality of subject software systems effectively. However, note that some components are un-testable with TestComplete (e.g., audio an installation components).

4.2 Debugger Applications

There are many desktop applications that are not designed to run with multiple instances on the same workstation. However, testing requires at least two instances to interact with one another (e.g., to make calls and send messages). Therefore, the tester needs to install and run multiple instances on the same workstation because the automated testing system can only access and perform actions on the applications on one single machine. These instances are known as “debuggers”. Debuggers facilitate testing of the application’s functionality. Components such as call controls, presence, and chat require multiple instances to interact with one another. Debuggers are the application instances that interact effectively with the tested application.

Call control is the most important and essential function for a telephony application. To demonstrate how debuggers work and interact with the tested application, we illustrate the testing of whether an outgoing call is successfully performed and whether the associated functionality works properly. In the following, we use the example of making a call to a debugger to explain how debuggers are used to facilitate testing.

First, the tested application instance makes a call to a debugger application. The debugger either answers or does not answer the call. We need to validate the different states: *ringing*, *talking*, or *being on hold*. Such functional requirements are validated by comparing phone iconic images and label texts in the tested product’s communications Window and communications Shutter. For example, to validate call status, phone icons can be checked; they should show *Ringing/Talking/On hold*. To validate the proper caller information, the caller information label

text shown in the communications Window and the communications Shutter can be compared with expected text. Call status can also be validated by checking the call status label text (i.e., *Ringing, Talking, On hold,* or *Held by*).

The status of an integrated application (e.g., Microsoft Live Communication Service, MSN, and Microsoft Outlook) can also be validated during a call by checking the corresponding iconic images. To ensure that a call is established successfully, one has to check the caller and the debugger call status. However, since an incoming call is also tested as a separate testing component, it is unnecessary to check the status of the debugger. In this example, only one debugger is needed. Multiple debuggers are used while testing components that involve multiple participants such as conference, multi-chat, and presence. Figure 4-1 depicts the interactions between multiple debuggers and the tested application. Appendix A lists sample source code implementing the test suites for testing Telephony feature.

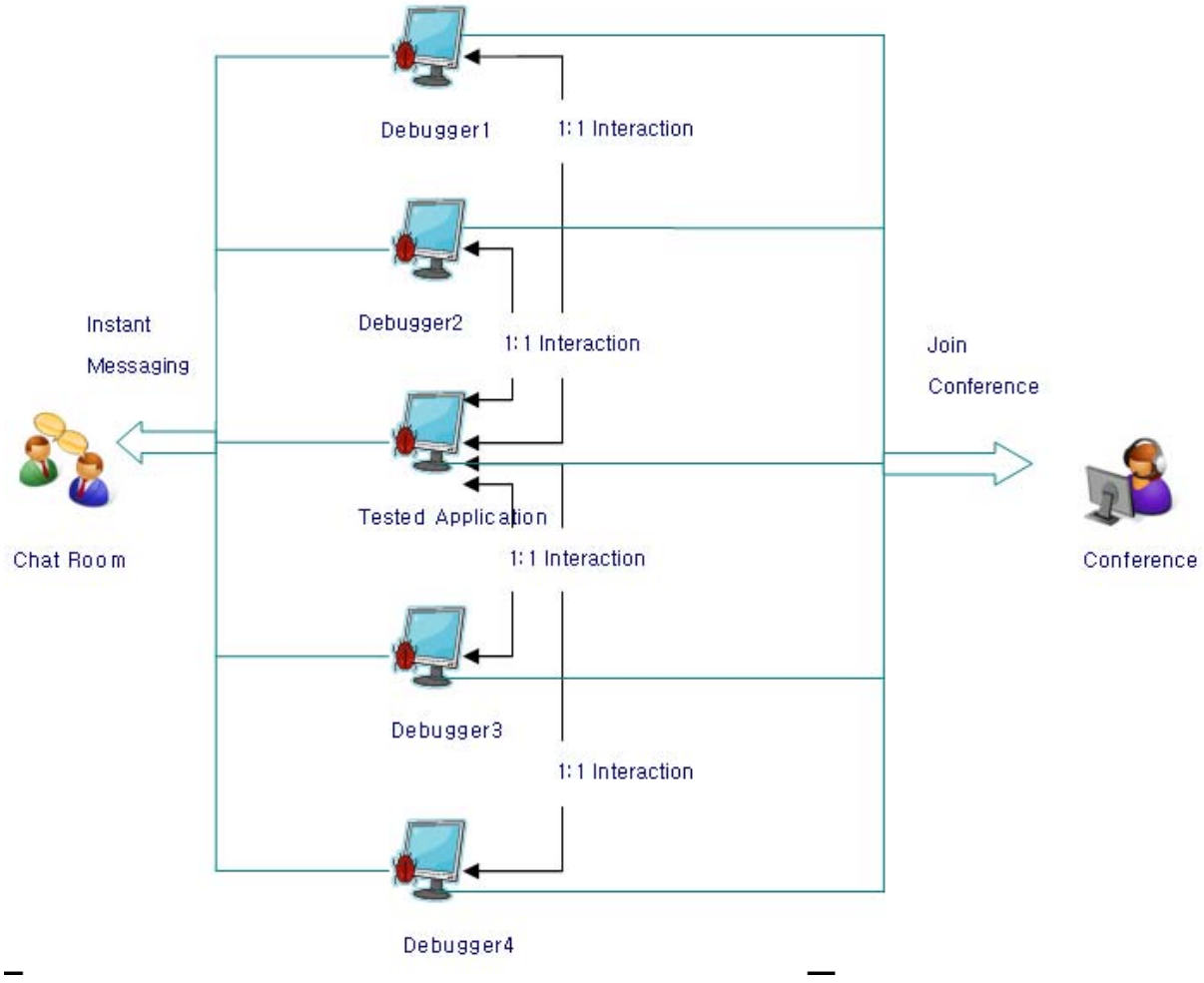

Figure 4-1: Interactions between Tested Application and Debuggers

4.3 Application Installation and Pre-Configuration

This section discusses the installation of the tested application and the debuggers. As mentioned earlier, the tested application is not designed for a user to install and run multiple

instances on the same workstation. As a result, to enable testing using debugger applications, special settings and configurations are needed for the tester to install and run multiple instances concurrently on the same workstation.

Multiple instances of Desktop Assistant typically do not run on the same machine. However, in order to perform test automation, multiple instances are required to run concurrently on one machine where the automation project is set up. To install debuggers, the user needs to make copies of the application file and the configuration file. The following steps install the necessary debuggers:

(1) Install the application being tested, Desktop Assistant, from the build directory. By default, all installation files will be stored in the installation folder (e.g., `C:\Program Files\NewHeights Software\DA-SIP`).

(2) In the installation folder, make three copies of the debugger `DA.exe` and call these files `BellPCMDebug1.exe`, `BellPCMDebug2.exe`, and `BellPCMDebug3.exe`. Further, make three copies of the configuration file `DA.exe.config` and rename these files `BellPCMDebug1.exe.config`, `BellPCMDebug2.exe.config`, and `BellPCMDebug3.exe. config`.

(3) Assign different port numbers to the different debuggers in the configuration files so that the different instances will run on different ports. For example, in `BellDebugger1.exe.config` file, assign `5061` as the port value for `Debugger1`. Figure 4-1 shows the configuration file after this change.

```
    <TrilliumStackSettings>
        <add key="TrilliumLog" value="" />
        <add key="Port" value="5061" />
    </TrillumStackSettings>
```

Figure 4-2: Assigning Port Numbers for Debuggers in Web Configuration

(4) In the configuration files, assign different names to the variable `MIProductName` (e.g., **`da1`** as depicted in Figure 4-2).

```
    <StartupSettings>
        <!-- use this value to disable the warning at startup about using
a dpi other than 96 -->
        <add key="UserWarn96Dpi" value="True" />
        <add key="ConfigAndLogName" value="da" />
        <add key="MIProductName" value="da1" />
    </StartupSettings>
```

Figure 4-3: Configuring Debuggers

(5) After the debuggers are installed, their application data paths need to be specified so that their corresponding data will be stored separately. Each instance should have its own data directory (i.e., `DA1`, `DA2`, and `DA3`) in the same directory as the path for the tested application data (e.g., `C:\Documents and Settings\izhou\Application Data\NewHeights`).

(6) To test Outlook integration, Outlook needs to be properly installed and configured before running the test automation software.

(7) MSN and LCS need to be integrated similarly for testing (e.g., MSN Messenger and Office Communicator need to be configured with a few on-line and off-line contacts in their contact lists).

4.4 Summary

The automated coverage depends on the objectives of the testing and the capability of the automation tool. In addition, choosing the right automated tool is extremely important. One needs to take into consideration the maintainability and reliability of an automated testing environment to achieve automation success. We have chosen TestComplete as our core testing automation tool.

Chapter 5
Testing Automation Case Study

The subject software system (i.e., the application software to be tested) for our case study is a VoIP (Voice Over Internet Protocol) telephony product of NewHeights Software Corporation called *Desktop Assistant*, which provides telephone service over a data network. It facilitates telephone and instant messaging (internal and external) communications for organizations. It also integrates readily with today's most popular desktop software applications, such as Windows Live Messenger, Microsoft Outlook, and Lotus Notes, to enhance their instant messaging, contact management, and telephone capabilities.

This chapter discusses the design and implementation of our testing automation system. We begin by analyzing the requirements of the tested subject system and discuss the features and support that the TestComplete environment provides.

5.1 Requirements Analysis

First, we analyze the requirements for implementing our automated testing system. Second, we analyze the goals for testing our subject system Desktop Assistant and then discuss the features of the automation tool TestComplete aiding the automated testing of Desktop Assistant.

Desktop Assistant was developed by NewHeights Software Corporation using the Microsoft Visual Studio .NET software development environment. Users can interact with this application through its user interface to utilize its functions effectively. Thus, it is critical that sufficient resources are devoted to the testing of the general user interface. Moreover, the application's functional requirements are tested by monitoring the user interface. The method for testing Desktop Assistant is basically regression testing. Since Desktop Assistant was developed in the Visual Studio .NET environment, it is a .NET application. Since our automation environment TestComplete supports testing of .NET applications, no compatibility problems arise between the tool and the subject system under test. Thus, the TestComplete's automated testing processes readily apply and work well for Desktop Assistant.

Testing the user interface of Desktop Assistant is a major undertaking and obviously a key component of its entire testing process. Users interact with the application effectively through a

GUI containing application windows, text labels, button identifiers, and iconic images. Figure 5-1 depicts an example Desktop Assistant window called *Communication Window*. The GUI components of this window requiring validation are the company logo, window title, button names, caller identification text, mute button, volume bar, close button, minimization button, and annotation button. Our test automation tool TestComplete includes special features and capabilities to simulate various user actions to test and exercise the windows and menus of an application, capture screenshots of user interface elements, and compare these screenshots with reference screenshots.

Figure 5-1: Snapshot of a Communications Window

To conduct and complete a test case, three stages must be executed:

(1) exercise the specific functionality,

(2) capture an actual result through exercising the functionality, and

(3) compare the actual behaviour with the expected behaviour.

To test a window, TestComplete can simulate user actions, such as a user calling another user, by producing a Communications Window. Thus, the effect of this action is that a Communications Window appears with buttons, text labels, iconic images, and a window title. To validate the generated graphical widgets in such a Communications Window, the generated window is compared to a pre-computed and stored image of a Communications Window [3, 25]. The sample source code for testing communications window GUI and its associated features is listed in Appendix B.

Since Desktop Assistant is a VoIP product, the basic functionality of this product includes operations, such as starting, connecting, and terminating a call, initiating a conference call, and transferring a call. Most of the functional testing can be carried out through the validation of text labels. For example, when making a call, the connection of the call can be validated by checking the communications status in the Communications Window above. If the call is ringing, the status text label will be *Ringing*; if the call is connected, the status will show *Talking*. Other labels, such as *On Hold* and *Conference*, are validated in a similar fashion. These text labels can be easily validated using the built-in functions `Verify()`, with two input parameters (i.e., condition and verification messages). For example:

```
Verify(sActualStatusLabel.indexOf (sExpectedStatusLabel) > -1,
“Call state expected to be” + sExpectedStatusLabel + , “Actual label
is” + sActualStatusLabel).
```

By comparing the actual status label captured at run-time and the expected status passed in through parameters, one can validate the state of the call. For instance, in the above example communication, the call is connected by showing the *Talking* status label. At run-time, the expression

```
sActualStatusLabel = w["callStateLabel"]["Text"]
```

evaluates to *Talking*, and one would pass-in

```
sExpectedLabel = "Talking"
```

and therefore the expression will evaluate to `True` indicating that this case passes (i.e., the function works as expected).

Longevity or *endurance testing*, evaluates a system’s ability to handle a constant, moderate workload continually [26]. Stability is obviously a major concern and hence significant testing effort is expended to exercise the functions of Desktop Assistant over long periods of time. For example, customers tend to make a call and talk for a long time, or make a large number of calls during a peak period. Automated testing systems ought to be able to handle this type of testing. For example, one can devise a test script so that a call can last for a long time.

One can also configure test scripts so that multiple calls can be made simultaneously. These longevity test cases can all be covered effectively by our test automation tool TestComplete.

5.2 Design of Testing Processes

After discussing selected testing requirements of our subject software product Desktop Assistant and selected applicable features of our test automation tool TestComplete, we now illustrate the design of the actual testing processes.

5.2.1 Desktop Assistant Feature Decomposition

The features that Desktop Assistant provides to customers include Voice-over Internet Protocol call, Chat or messaging, MSN integration, PIM integration, and Contact management. These are the top-level features. As a result, testing and quality assurance of these features is extremely important. Each major feature can be broken down into sub-features. Similarly, the testing problem can be decomposed into testing sub-features individually.

Testing VoIP involves call status, call history, call record, call forwarding, call transfer, and conference. Testing these functions thoroughly and testing the integration of these sub-features ensures the quality of the upper level features. In this case study, we implemented

different regression test suites for the different sub-features. Of course, the quality assurance for a particular feature depends on the design and implementation of its regression test suite. Running and passing the well designed regression suite for a particular feature means that the quality aspects addressed this suite are assured.

Chat status, chat history, presence, and file transfer constitute the features of instant messaging. MSN integration deals with MSN presence and MSN chat function. PIM integration is decomposed into Outlook integration and Lotus Notes integration. Finally, contact management consists of corporate contact, personal contact, contact presence, and contact groups. Test suites are designed appropriately to address the testing of the individual features. Figure 5-2 depicts a decomposition of the Desktop Assistant features.

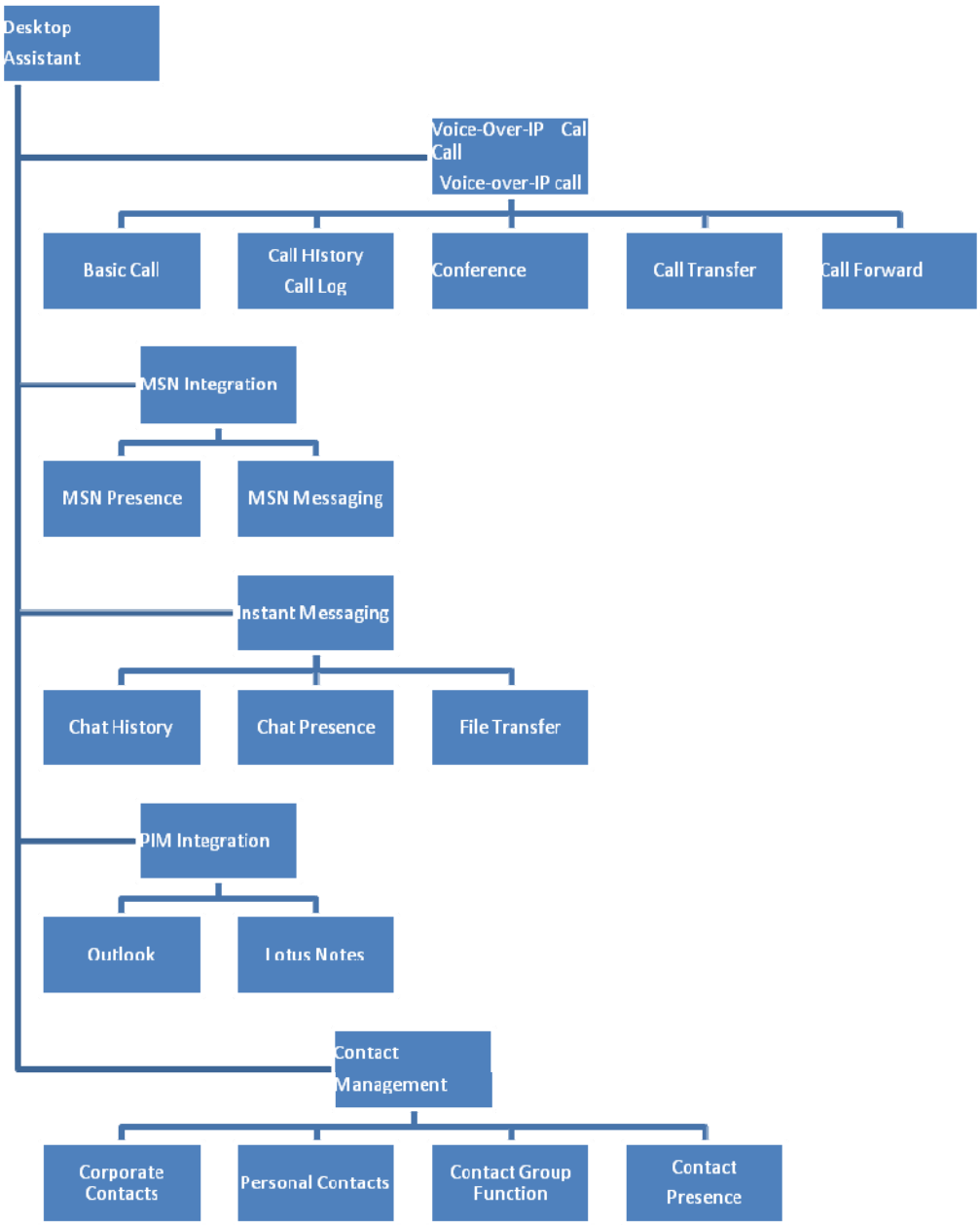

Figure 5-2: Desktop Assistant Feature Decomposition

5.2.2 Architecture of Testing Automation System

In addition to understanding the tested product features, one must also take into consideration the efficiency and maintainability of the automated testing system. Testing flow requires consistent design and control to achieve high efficiency, and maintainability can largely be attained by effective code reuse. The two characteristics, testing flow and code reuse, of the automated testing system are discussed below in more detail.

When designing an automated testing system, one needs to take into consideration adoption of the automation system and in particular how a person is to be trained to use the software effectively. For example, a user needs to start up the application and log in to access the product features and, after finishing using the application, one needs to shut down the application. Thus, to test one feature, one always starts up, logs in, and shuts down the application. As a result, the start-up, login, and shut-down can be designed as reusable and maintainable test suites. Another reuse example arises when testing the basic call, call history, or conference functions (i.e., a call function can be reused).

We can produce a consistent design to guarantee efficiency and maintainability by introducing the following components: start up, shut down, login, and logout contain the corresponding functions; call utilities, which lists a set of call functions that any testing call

feature needs to access; tools, which provide a sequence of validation functions, such as text and image validation; as well as shutter utilities, which provide a set of functions to navigate the shutters.

Application debuggers are also introduced to aid the testing of call and chatting functions. Figure 5-3 illustrates the architecture of the overall testing automation system.

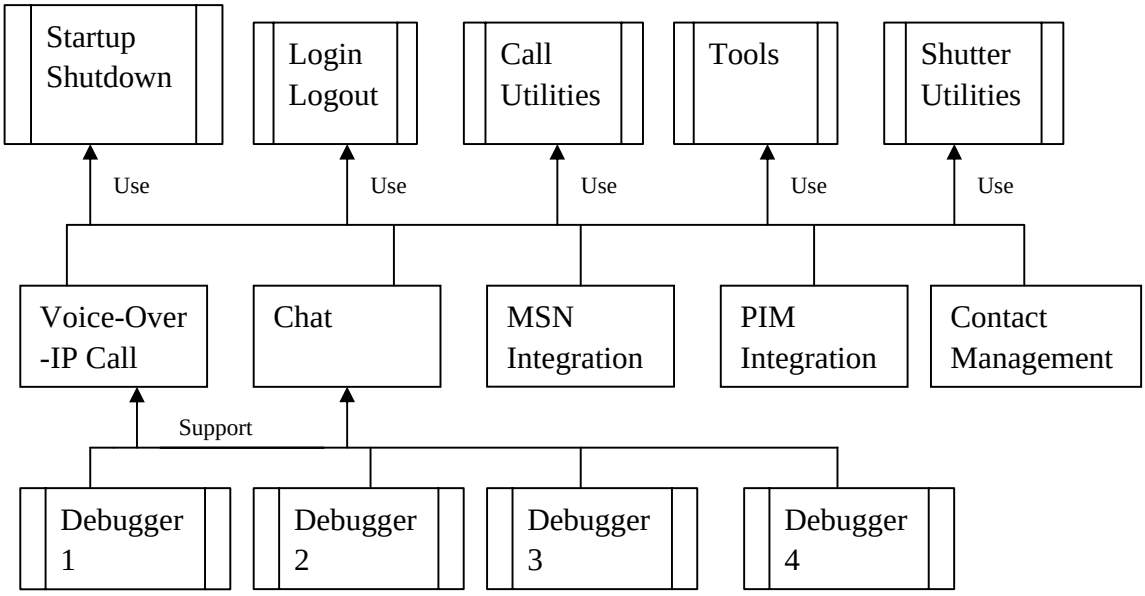

Figure 5-3: Architecture of Testing Automation System

5.2.3 Functional Design

We now analyze the functions to be tested and design the system accordingly. Assuming that a test case has been written to ensure that the basic call functionality works as expected, we

now illustrate how this case is dealt with throughout the automation project. Suppose the input for this test case is: “a client using the tested application receives a call from another client using a debugger instance of the application”; the resulting output for this test case is: “the call goes through and is connected properly.” To run this test case, the test suite *Basic Call* in Figure 5-2 located within *VoIP Call* in Figure 5-3 needs to take the input and produce an output, and then validate the output and generate a result.

The input assumes a set of preconditions as follows: start up the application; start up one of the debuggers; log into the application; and log into the debugger—these are the functions predefined in the *Startup*, *Shutdown*, *Login*, *Logout*, and *Debugger1* components depicted in Figure 5-3 and referred to as *Basic Call* component, which belongs to the *VoIP Call* component in Figure 5-2. After the preconditions are satisfied, *Debugger1* makes a call, and the *Basic Call* suite calls the function *AnswerCall* defined in *Call Utility*. The call is now supposed to be connected. Consequently, an output appears. *Basic Call* then calls a verification function defined in *Tools* to validate the output and generate a test result. At the end, both the tested application and the debugger need to be shut down and finish running the test suite *Basic Call*.

It is usually impractical to run only one test case at a time. One test suite usually addresses a large number of test cases and several test suites can be grouped together to run at once. Therefore, the structure, control, and orchestration of the test suites are very important.

5.2.4 Deployment of Automated Testing System

This section discusses how the system is deployed and used after implementation. It is important for the test engineers to organize and manage the test suites well. As a tester designs and writes test plans prior to implementation of the automated system, it is best to organize the test suites implemented in the system according to the test plans.

A test suite in the automation system is designed to test a number of functions that the tested software offers. It usually returns the best practical result if a test suite acts according to a test plan written in reflection of the functional requirement specification. All the test cases in one test plan can then be grouped to run as a whole. Figure 5-4 displays a test suites run.

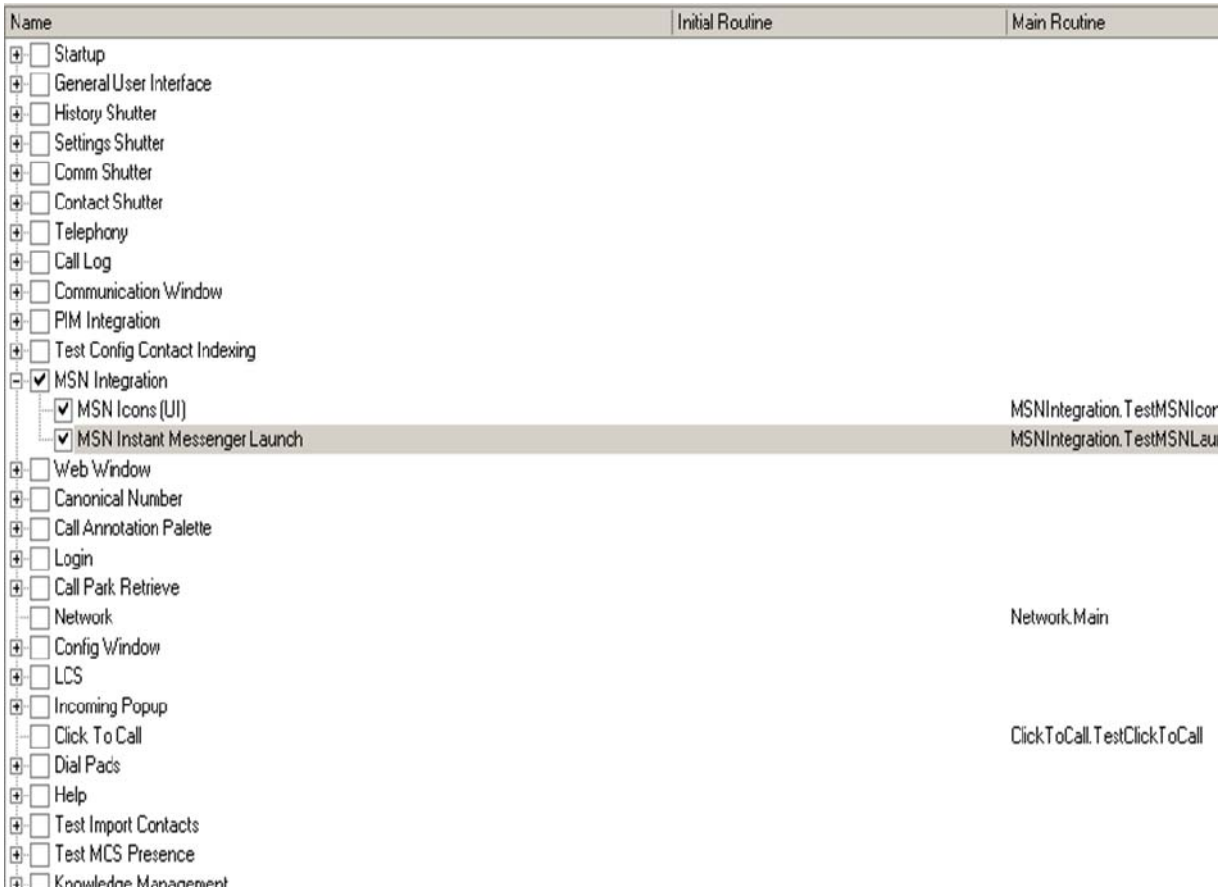

Figure 5-4: Snapshot of Test Suite Display

In practice, a test plan is designed to correspond to part of a requirements specification written for a feature required by a customer. Each feature is decomposed into sub-features. Accordingly, a test suite contains functions that address the different sections and sub-sections of a test plan. Table 5-1 shows the relationships among feature specification, test plan, and test suite.

Table 5-1: Feature Specification, Test Plan, and Test Suite

Feature Requirement Specification: MSN Integration	Test Plan Name: MSN	Test Suite Name: MSN Integration
MSN Icon Display	GUI testing	MSN Icon (UI)
MSN Launch	Launch msn function testing	MSN Instant Messenger Launch

5.2.5 Estimation of Test Effort Saved By Using Automated Testing Engine

This section analyzes the automation test case coverage and time saved on regression and longevity testing by using the automated testing engine.

The test plans written according to the requirements specifications for the subject software, Desktop Assistant, include the following features: Voice-over-IP Call, MSN Integration, Instant Messaging, PIM Integration, and Contact Management. These test plans were designed and written for manual testing. Automated test engineers need to go through the test plans and take out all the test cases that can be automated and assemble them into new test plans. Then they update the new test plans for automation test.

The following table indicates the number of test cases in total and number of test cases automated for each feature.

Table 5-2: Automation Coverage Statistics

Features	Total Test Cases	Automated Test Cases	Percentage
Voice-over-IP Call	1,500	500	33.3%
MSN Integration	400	250	62.5%
Instant Messaging	800	350	43.7%
PIM Integration	1,200	450	37.5%
Contact Management	600	180	30.0%

From the above table, we can compute the automation coverage. This percentage approximately reflects the test time saved on regression testing and longevity testing. If we subtract the time spent maintaining the engine, organizing automation testing, and collecting the test result reports, then we obtain slightly different figures. Figure 5-5 depicts this estimation on testing effort saved using the automation engine.

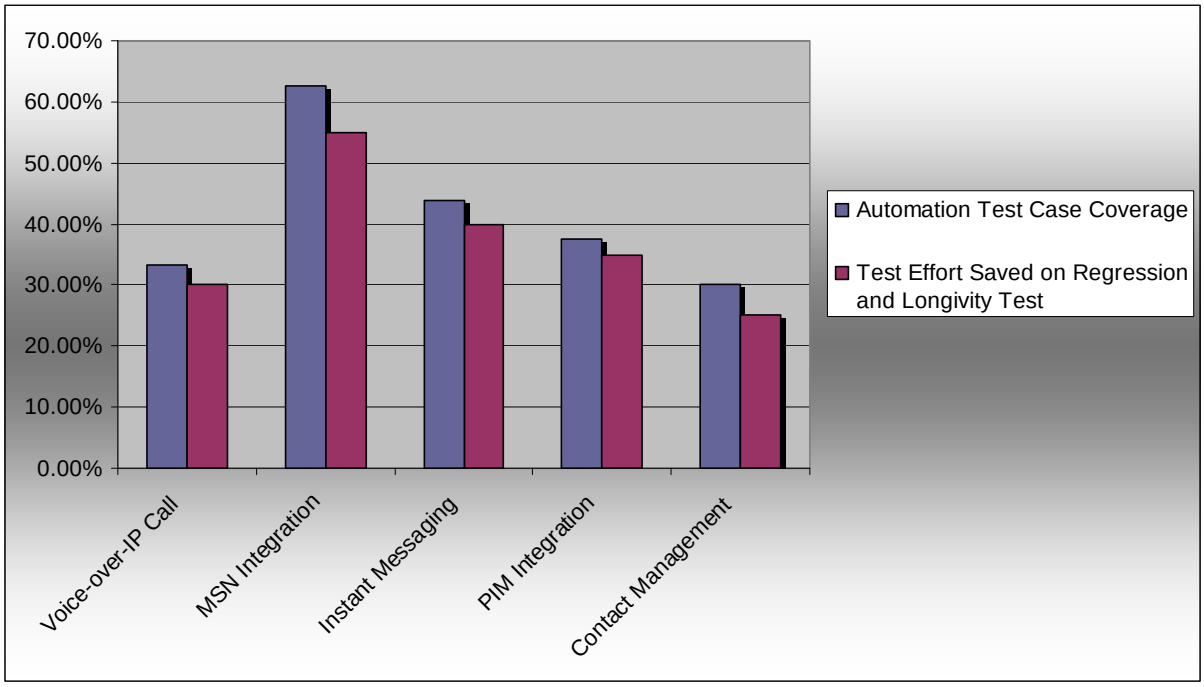

Figure 5-5: Estimation of Test Effort Saved using Automated Test Engine

5.3 Implementation of Automated Testing Engine

This section describes the implementation of our automated testing engine and associated testing processes for Desktop Assistant. The implementation of an automated testing engine for a subject system consists of three major parts: (1) identification and implementation of reusable infrastructure components; (2) development of test suites corresponding to test plans; and (3) development of scripts to orchestrate and manage the execution of run test suites.

5.3.1 Reusable Infrastructure Components

Reusable infrastructure components provide basic functionality for feature test components as outlined above.

Example infrastructure components are the Start-up and Shutdown components, which are used by many subsystems. However, these components are not only infrastructure components for most test suites, but also test suites themselves thereby ensuring that the start-up and shutdown functions of the subject system work properly. Therefore, these components need to validate these functions over and above providing functionality for other test suites.

Figures 5-5 and 5-6 depict the sample code for the `Startup()` and `Shutdown()` functions, respectively. The start-up process involves the function of launching an application process (i.e., `LaunchP()`) and involves validation. For instance, it can detect whether the software is running; if so, it will shut down the application and then re-start it. The `Shutdown()` function is simpler than the `Startup()` function. It closes the application and verifies that the application can be shut down properly.

```
 function Startup(bFirstTime, isPIMInvolved) {
    InitializedLocalVariables();
    if(bFirstTime == null) {bFirstTime = false}
    var p = GetP(0);
        Assert(p["Exists"] == false, "Startup(), the program is already
active! The " + sTestedProductName + " process must be completely shut down
before running test suites.");
    if(p["Exists"]){
        Shutdown();
        p = GetP(0);
        Assert(p["Exists"] == false, "Startup() - Main process is already
running! Shutdown must be called first!");
    }
    if (bFirstTime) {DeleteConfigFiles();}
    LaunchP();
    PollForProcess( sTestedProductName );
    if (bFirstTime){
        ConfigAudioSettings(isPIMInvolved);
        ConfigFirstTimeLogin();
        VerifyDefault();
        VerifyLoginStatus( "Loged in as", null);
    } else
        SecondTimeStartupAndVerifyCorrectSettings();
 var w = PollForWindows(sTestedProductName, "MainWindow");
 w["Activate"]();
 Assert(w["Exists"] == true, "FirstRun() - main window doesn't exist.");
 sys.Delay(3000);
}
```

Figure 5-6: Application Startup Function

```
function Shutdown(firstTime) {
if(firstTime == null)
       Exit();
else
       Exit(1);
var p = PollForProcessTermination(sTestedProductName);
Assert(p["Exists"] == false, "Shutdown() failed. Main process is still
alive!");
```

Figure 5-7: Application Shutdown Function

The Login and Logout components are implemented in the same manner as the Start-up and Shutdown components. `Login()` is defined to log into the application. Whenever features need verification in on-line mode, this function is invoked. It can also work as a stand-alone test case to validate the login functionality of the subject software system. Similarly, the function `Logout()` can be invoked when application features need verification while in off-line mode. In addition, it can validate the logout functionality.

```
function Login() {
	Log["Message"]("Login()");
	MainMenuClick("[0|0]");
	Sys["Delay"](1000);
	VerifyLoginStatus("Logged in as");
}

function Logout() {
	Log["Message"]("Logout()");
	MainMenuClick("[0|1]");
	Sys["Delay"](1000);
	VerifyLogoutStatus("Logged Out");
}
```

Figure 5-8: Application Login and Log Functions

In addition to the above infrastructure components, the Call Utility infrastructure component supports the testing of VoIP call functionality. It provides a list of functions such as `MakeCall()`, `AnswerCall()`, `TransferCall()`, `Conference()`, and `ForwardCall()` which used repeatedly in various test suites. Another essential unit of infrastructure components is the set of Shutter Utilities. Shutter Utilities define the functions to handle shutter activities. For instance, a shutter needs to be undocked for itself to be tested as a separate window. Shutter Utilities provide the function `UndockShutter()` by passing the shutter name.

Finally, the Tools unit defines miscellaneous utility functions. Basically, this unit contains three categories of functions: functions that activate processes or windows (e.g., `LaunchP()`—launches the application process and `GetW()`—brings the window into focus); functions that validate objects (e.g., image, text, and label validation); and functions that interact with windows (e.g., maximizing, minimizing, and closing windows).

We also define application debuggers to support testing of the main functionality. Application debuggers are instances of the tested subject application which interact with the tested application during call and instant messaging. Only supporting roles which do not involve validation are assigned to application debuggers. For instance, a debugger can make a call to the tested application to ascertain that the receipt of call function works well for the tested application. Another illustrative example is when the subject application invites a set of, say, four debuggers to a conference call. When the conference is initiated from the tested application, each debugger receives a call and answers it. After all these actions are performed, the connection of the conference can be validated. Figure 5-8 depicts the test script for a conference call.

```
function TestConference() {
	Startup(1);
	BellDebugger1Startup();
	BellDebugger2Startup();
	BellDebugger3Startup();
	BellDebugger4Startup();
	MakeCall(sBellDebugger1Address, 1);
	BellDebugger1AnswerCall();
	Conference(sBellDebugger2Address);
	BellDebugger2AnswerCall();
	Conference(sBellDebugger3Address);
	BellDebugger3AnswerCall();
	Conference(sBellDebugger4Address);
	BellDebugger4AnswerCall();
	VerifyCommunicationStatus(false, "Conference");
}
```

Figure 5-9: Conference Call Test Script

5.3.2 Feature Test Components Implementation

This section discusses the implementation of feature test components. Feature test components validate feature requirements of the Desktop Assistant application. Test plans have been written according to the feature requirements specification. As a result, the written test

plans can guide test engineers in the implementation of the feature test components. Since a test suite is implemented in conjunction with the corresponding test plan, it is important to design a consistent test plan to be able run the test suite effectively.

We use the MSN Integration mentioned in Section 5.2.3 as an example to illustrate the implementation of the MSN test suite. The test plan consists of two sections: GUI and functionality testing. Table 5-2 depicts the MSN test plan designed for automated testing. For the GUI section, the automatable steps are:

(1) open Windows Messenger,

(2) open MSN Messenger, and

(3) invoke the application.

The verifiable results are MSN presence appears within the application window and MSN presence does not appear within the application window. For the second section, the automatable step is click messenger icon and context menu when on-line and off-line. The verifiable results are instant messaging window appears, warning message appears, and caller information appears.

Table 5-3: MSN Test Plan

Section 1: GUI Testing	
Input	**Expected Output**
1. Start the application 2. Create a contact with valid MSN info	MSM presence should be visible in the application
1. Create a contact with invalid MSN address	MSN presence should not be visible in the application
1. Create a contact with email address only (with no MSN info)	MSN presence should not be visible in the application
Section 2: Functionality Testing	
Input	**Expected Output**
1. Click on the Messenger icon for a user who is currently online	Should open a Messenger chat window with the user who's icon was selected
1. Click on the Messenger icon in the Communication Window for a user who is currently offline 2. Click again	Should only pop one modal warning
1. Open a Messenger conversation via the context menu with an application/MSN contact	Same as clicking icon
1. Receive an incoming call with the Messenger match and personal contact match	The caller name is shown as the one in Personal contacts. Status is shown as in Messenger

After designing a test plan, it is reasonably straightforward to implement the test suite. For each test case, we need to write a test script. Figure 5-9 depicts the test script for GUI testing. Similarly, the functional testing section of the test plan can be mapped to the code shown in Figure 5-10 (See Appendix C for more details).

```
function TestMSNIcon(){
    var sPrefix = "ImagesUsedToCompare\\MSN\\";
    var sPostfix = ".bmp";
    Startup(1);
    UndockShutter("Contact");
    var p = Sys["Process"]("DA");

    CreateContactWithMSNOnly("Contact with MSN Only", sCompanyName, "MSN
address 1", sValidMSN);
    Sys.Delay(2000);
    w = GetW("ContactShutter");
    w["Activate"]();
    oIcon = w["WaitChild"]("icon0", 1000);
    Verify( oIcon["Exists"], "Contact with a valid MSN address correctly
shows MSN icon" );
    DeleteFirstContact( "msn" );

    CreateContactWithMSNOnly("Contact with MSN Only", sCompanyName, "MSN
address 1", "invalidMSN@newheights", true);
    var p = Sys["Process"]("DA");
    w = p["WaitChild"]("MessageBoxForm", 1000);
    Verify(w["Exists"], "Warning message about non existing MSN address
appears.");
    w["textLabel"]["Drag"](124, 2, 341, 3);
    w["button0"]["Click"](28, 9);
    w = GetW("ContactInformationWindow");
    w["Activate"]();
    w["okButton"]["Click"](52, 16);
    Sys.Delay(2000);
    w = GetW("ContactShutter");
```

```
    w["Activate"]();
    w["contactLabel"]["Click"](37, 9);
    filename = sPrefix + "FocusedGroup1" + sPostfix;
    var newfilename = sPrefix + "New_InvalidMSN_Without_MSN" + sPostfix;
    SaveImage(w["contactLabel"], newfilename);
    Verify(CompareImage(newfilename, filename), "Focused group 1 icon is
correct.");
    oIcon = w["WaitChild"]("icon0", 1000);
    Verify( oIcon["Exists"], "Contact with and invalid MSN address does not
show MSN icon" );
    DeleteFirstContact( "none" );


   CreateContactWithEmailOnly("Contact with Email Only", sCompanyName,
"Work Email", "emailonly@newheights.com" );
    Sys.Delay(2000);
    w = GetW("ContactShutter");
    w["Activate"]();
    oIcon = w["WaitChild"]("icon0", 1000);
    Verify( oIcon["Exists"], "Contact without an MSN entry does not display
the MSN icon" );
    DeleteFirstContact( "email" );
    DockShutter("Contact");
    Shutdown();
}
```

Figure 5-10: MSN GUI Test Script

```
function TestMSNLaunch(){
    ...
   CreateContactWithMSNOnly("Contact with MSN Only", sCompanyName, "MSN
address 1", sValidMSN);
```

```
    Sys.Delay(2000);
    w = GetW("ContactShutter");
    oIcon = w["WaitChild"]("icon0", 1000);
    Verify( oIcon["Exists"], "Contact with a valid MSN address correctly
shows MSN icon" );
    w["icon0"]["DblClick"](5, 8);
    VerifyIMWindowExists();
    CloseIMWindow();

    w = GetW("ContactShutter");
    w["contactLabel"]["ClickR"](38, 9);
    w["PopupMenu"]["Click"]("[0]");
    VerifyIMWindowExists();
    CloseIMWindow();

    CreateContactWithAllData("Contact with All Data", sCompanyName,
                             "P1", sBellDebugger1Address,
                             "E1", sValidEmail,
                             "MSN1", sValidMSN );
    BellDebugger1MakeCall( sTestedProductAddress, 1 );
    Sys.Delay(2000);
    ClickOnIncomingPopup();
    VerifyIMWindowExists();
    CloseIMWindow();

    AnswerCall();
    Sys.Delay(1000);
    ...
    ClickOnCommWindow();
    VerifyIMWindowExists();
```

```
    CloseIMWindow();
    w = GetW("CommunicationWindow");
    w["infoLabel"]["ClickR"](94, 7);
    w["PopupMenu"]["Click"]("[1]");
    VerifyIMWindowExists();
    CloseIMWindow();

    w = GetW("MainWindow");
    w = GetW("ContactShutter");
    w["contactLabel"]["ClickR"](22, 11);
    w["PopupMenu"]["Click"]("[2]");
    VerifyIMWindowExists();
    CloseIMWindow();
    ...
```

Figure 5-11: Test Script for Validating MSN Integration Functionality

The MSN Integration test suite is an example illustrating the implementation of the automated testing system in terms of test suites. All other product features mentioned in Section 5.2.1 can also be implemented in the same manner. Test plans are designed and written before implementing the test suites. Then each test case is mapped to test script in the corresponding test suite.

For example, we can use the reception of a call to demonstrate the error handling method. When a user receives a call, the incoming call popup window is supposed to appear within a reasonable amount of time. In the program, one specifies the system delay time to allow the

programs to find the window. However, when the network is abnormally slow, it takes a different amount of time for the window to show up. When the automation programs attempt to detect the window and try to perform other activities on the window, but cannot find it, they will stop running. In this case, one must insert an error-handling method in the program to pass the error state and go to the next test case. Figure 3-1 depicts a sample test script to handle abnormal situations.

As mentioned in Chapter 3, the error handing and system recovery methods are extremely important to an automated testing system. We continue with the example in Chapter 3. When a user receives a call, the incoming popup can take a long time to appear due to network stability issue. If the network is stable, then the incoming popup will show within five seconds. Instead of setting the time to five seconds, one inserts a loop into the code and sets the time to one minute for the system to detect the window's existence. As soon as the system detects the window, it continues immediately with the execution of the testing script. Otherwise, if the window does not appear within the one minute timeframe, the system will terminate the function and continue with other test cases. Figure 5-11 depicts the error handling code of the test script.

```
function AnswerCall() {
 Log.Message(“AnswerCall()”);
 var w;
 p = GetP();
 w = p["WaitChild"]( "TelephonyPopupWindow", 1000 );
 var i = 0;
 while ( i < 60 && !w["Exists"]) {
        Sys["Delay"](1000);  // wait a second then try again
        p = GetP();
        w = p["WaitChild"]( "TelephonyPopupWindow", 1000 );
        i = i+1;
 }
 w = GetW("TelephonyPopupWindow");
 w["Activate"]();
 w["answerButton"]["Click"](41, 8);
 Sys["Delay"](1000);
}
```

Figure 5-12: Error Handling Test Script

5.3.3 Testing Flow Control

This section illustrates the execution of feature test components. The order of executing test components is important. It is crucial for test engineers to organize and control the testing flow efficiently. After all the components are in place in terms of coding, the test plans designed

for automated testing should have full coverage. It is important to orchestrate the execution of the test suites.

As mentioned in Section 5.2.3, test suites can be run together all at once or individually. Test flow can be controlled in *Test Suite Panel* of TestComplete, as shown in Figure 5-12. When selecting multiple test suites to run, it begins by executing the selected test suites from top to bottom. The test suites can also be expanded and its internal functions can be selected or deselected individually. Since functions from test suites are not necessarily run at once for a test suite, it is important for them to be implemented in a way that they can run individually.

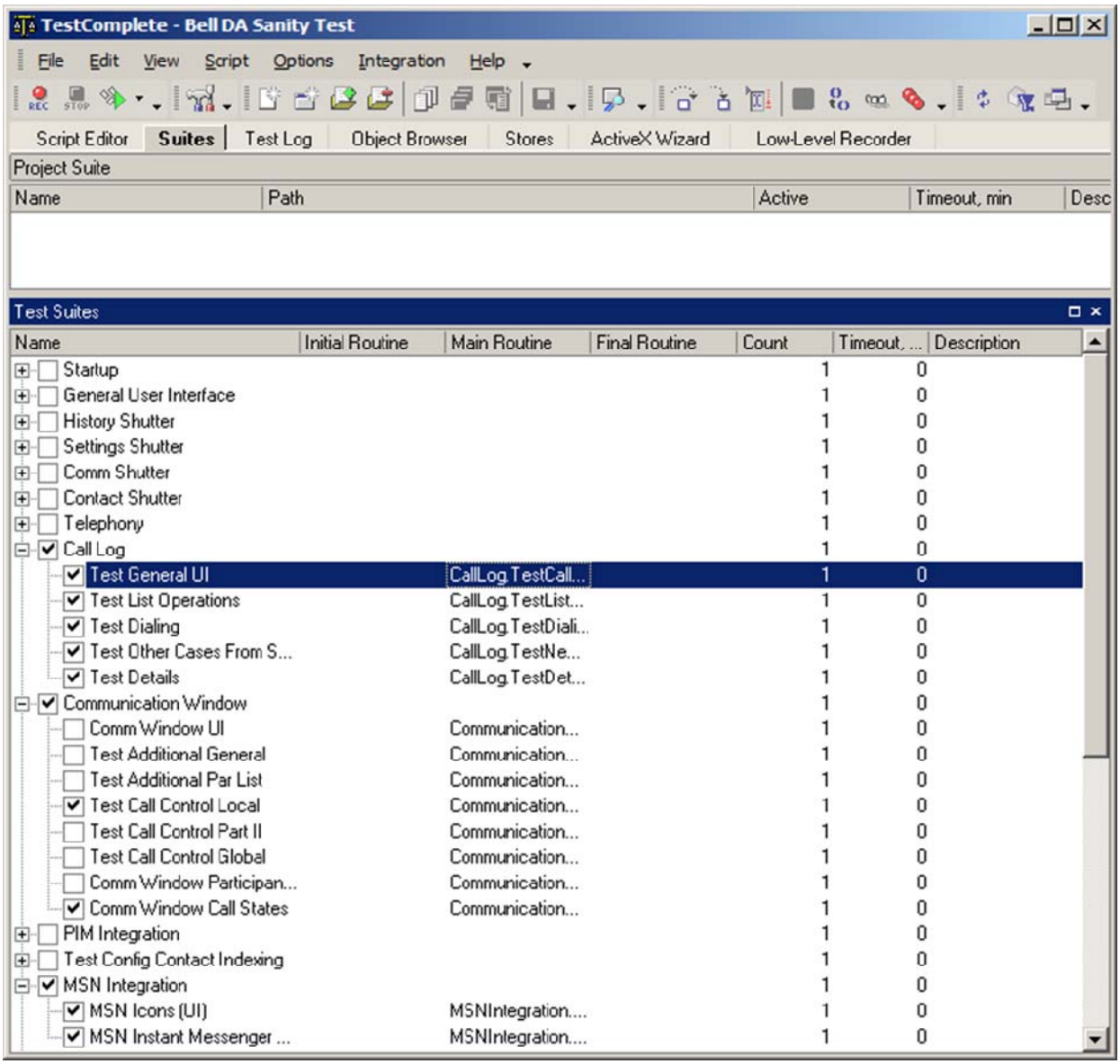

Figure 5-13: Test Suite Selection

From a testing point of view, regression testing is usually appropriate for areas with high risks. However, it is useful to implement test suites so that partial regression can be performed. The example in Figure 5-12 indicates that a round of regression testing on the feature *Call*

Record and on some functions from the communication window and MSN features are to be performed.

5.4 Summary

This chapter introduced the design rationale and selected implementation details of the automation engine. Based on the supporting features of TestComplete and the requirements of the tested application, we designed the architecture of the automated testing engine and processes.

Chapter 6
Evaluation

The goal of this research was to develop methods and tools to automate the testing processes of communications applications. To evaluate our methods and tools we used the commercial application Desktop Assistant as a case study. The assumption was that if our automation approach proves valuable and successful for this typical communications subject system, then the approach will likely be applicable and generalize to other applications.

In Chapter 3, where we discussed the requirements for our automated testing system, we already started to discuss quality criteria such as maintainability, reliability, and usability. This chapter summarizes these findings further and discusses other non-functional requirements, such as the ability to recover from error and high test coverage. We also document the benefits and limitations of our automation approach to testing and compare it to manual testing. The chapter

concludes by discussing the experience and lessons learned in developing our automated testing system.

6.1 Assessing the Requirements for an Automated Testing

As outlined in Chapter 3, ideally an automated testing system should be maintainable, reusable, reliable, and user friendly.

Maintainability and reusability issues: Several reusable and easy-to-maintain units have been implemented as part of the automated testing system aiming to improve its maintainability with respect to changes in the testing processes as well as changes in subject systems. Our subject system, Desktop Assistant, was an excellent case study because it changed frequently while we developed our automated testing system. The tested software keeps changing in response to customer requirements (e.g., over time features are added and the GUI changes in response to newly added features). It is costly to add new test units and to change the existing test units to accommodate such changes in the tested software. Therefore, we designed our test automation system to include reusable and easy-to-maintain components. For instance, when our subject software system added the new feature *caller ID*, the caller identification appeared in a variety of functions such as call transfer, conference call, and call forwarding, or even instant messaging. Without the reusable component, *CallUtilities*, one would have had to update all those components involving caller identification to verify that caller ID was implemented

poperly in the subject system. Since our automated testing system includes a *CallUtilities* unit, one only need changing the *call* function in *CallUtilities* and the *caller ID* function to test all related components effectively.

Reliability issues: Of course it is extremely important that the automated testing system be reliable. If we cannot assume that it is reliable, we will have difficulties deciding when a problem arises whether it is in the subject system or in the testing environment. Figuring out the root cause of the problem (i.e., subject system or testing environment) is likely very time-consuming. Our automation system eliminates possible problems of this sort by introducing exception handling capabilities to capture faults and minimize false alarms. Thus, errors are captured and recognized and not hidden, masked, or ignored due to the automation process.

Usability issues: If an automated testing system is difficult to use or if the automated testing process takes longer than manual testing, it is not very useful. TestComplete provides a user-friendly environment for users to set up and run testing projects effectively. When designing and implementing our automated testing system, usability was a key concern. We addressed this issue by providing integration capabilities (e.g., ease of integration of existing test plans) to allow the introduction of the automation process incrementally (i.e., testers can manage the test suites according to test plans written previously). This strategy will hopefully ease the adoption of the automated testing system significantly. In addition, the error reporting scheme was

designed carefully to ease comprehension for testers and thus reduce the time it takes to comprehend and localize a problem or a fault.

Using data-mapping methods, a test case can be mapped into a test function in our automated testing approach and thereby characterize test coverage. In a test plan spreadsheet, one can mark test cases for coverage by the automated testing system and thus exclude them from manual testing. However, as mentioned earlier, not all test cases can be covered by automated testing system processes. One important limitation is that our automated system cannot handle audio testing. Since Desktop Assistant is a VoIP telephone system, audio functionality is clearly an important area to test. Other automation tools such as automated SIP protocol conformance test tool can test audio by validating Session Initiation Protocol (SIP) address transformations.

An important feature of an automated system is that it can run unattended. It should run automatically for a certain period once it is set up properly and invoked. Therefore, it is important for an automated system to recover when errors occur. Our automated system includes an error recovery method when a function is likely to halt. For example, when a function causes the software to stop functioning, the system first detects whether the software is no longer running properly and then reports the error; then the system can recover from such an error by halting the process and starting it again. However, not all problems that cause the system to stop running can be recovered in this way. There is no easy way to deal with the problem

“application not responding.” When the tested application stops, the automation system has no way to determine whether it is experiencing an application problem. Consequently, the continuity of the execution will be broken. The tester has to step in and stop the execution of the application and start it again at the spot where it had stopped working.

6.2 Benefits and Limitations of Test Automation

Conceptually, it is satisfying to have computing systems detect and report (and perhaps even correct) their own bugs automatically [13]. However, that expectation is unrealistic at the current stage of computing technology. Yet, test automation constitutes an important step in that direction. Compared to manual testing, automated testing has a number of benefits and but also some limitations.

Typically, automated testing improves the testing efficiency and may reduce testing costs significantly. After designing and implementing our automated testing system, we put it into practice at NewHeights Software Corporation. We can certainly attest that using this system for testing saves a great deal of testing effort. Most regression testing can be automated and accomplished using our automated testing system. With today’s resource limitations, automation is critical before product releases. Moreover, the approach proved to be reliable and user-friendly. It has been implemented so that various testers can control and run it. The test records are clear

and easy to understand. Since the automated testing system can run unattended, it frees up resources to perform advanced manual tests. In addition, TestComplete has access to certain system parameters invisible to testers. Hence, it is easier for our automated testing system to reach hidden areas of the tested software than for human testers during pure manual testing. Overall, automated testing has improved the testing process at NewHeights Software Corporation significantly.

However, one should not forget that automated testing is an extension of or complementary to manual testing. It should not be regarded as a replacement for manual testing. Automated testing has significant limitations. First, it cannot test functions that have no expected pass criteria (i.e., no criteria can be fed into an automation system as the expected results). Second, automated testing requires human interaction, although it can run unattended. It requires testers to set up parameters such as test input and expected results before starting execution. During operation, if the application halts due to some functional error, it cannot recover until testers stop the process and restart the execution of the application. Third, automated testing scripts are painful to maintain when software frequently changes. Accordingly, it is not recommended to use automated testing in a development process where new features are added in short order. One needs to analyze testing requirements carefully to make the decision to employ automated testing.

6.3 Experience and Lessons Learned

This section describes and analyzes the experience gained and lessons learned while implementing and using the automated testing system [19]. By analyzing the test requirements and investigating automation methods and tools, we have implemented the automation system with easy-to-maintain and code-reuse concepts in mind. It can be rather painful to maintain an automated system as tested software keeps changing over time to satisfy customers' needs. The entire automation system may fail in the adoption process or potentially fall into disuse because of maintenance problems. However, so far our automation system has not encountered this problem. After analyzing the test requirements, we discovered that the GUI of the subject software does not change frequently. We also learned that majority of its fundamental functions do not change. Even when adding features, those fundamental functions can easily be modified to test the changes.

False alarms are a key concern and can frustrate automated testers. The accuracy of test results is usually affected by false alarms. If exceptional cases cannot be caught, then the system sends false alarms. We have basically eliminated false alarms in our automated system by strategically inserting a set of exception handling methods into the code.

In the early stages of our automated testing tool, more time was spent on automating test scripts than on actual testing. However, in our automation experience, we discovered that planning is more important than implementing the automation scripts. Ignoring the test requirements can obfuscate the real goal of automated testing, which is to test the subject application thoroughly. Therefore, carefully analyzing test requirements and marking what test cases can be integrated into automated testing becomes crucial.

Another lesson we learned from developing the automated testing system is that not all testers should be involved in implementing such a system. Only those testers with software development experience should be involved. It requires a solid understanding of software engineering principles as it involves code maintenance and reuse. The scripts generated by an inexperienced test engineer can break the structure of the system and, thus, affect testing adversely. Although manual testers should not be involved in implementing the system initially, they should be trained to understand the automated testing system completely to be able to use the system effectively and generate simple testing scripts when they are more familiar with the system.

6.4 Summary

Given our experience in designing, implementing, and using the automated testing system, we are confident that the system satisfies the test requirements as outlined in Chapter 3. The system is maintainable, reliable, and user-friendly. It covers many test areas and improves regression testing significantly. It also handles error exceptions effectively. However, some limitations remain and will persist since automated testing will never completely replace manual testing. Further investigation and application of our automated testing system in industrial practice is certainly warranted.

Chapter 7
Conclusions

7.1 Summary

As software complexity increases, it becomes exceedingly important and urgent for software test engineers to find means to reduce their testing effort. With the evolution of software development processes and tools, programmers have become much more efficient and prolific in producing code. Consequently, software testers are under considerable pressure to test an increasing amount of code more efficiently and more effectively. Test automation is a way to alleviate this pressure and be better prepared for the onslaught of old and new code to be tested.

This thesis presented the design and implementation of an automated testing system based on the automated testing tool TestComplete. The automated testing system and its associated testing processes were validated with an industrial case study. To illustrate the

application and deployment of our testing system we employed the commercial product Desktop Assistant. Our automation tool and its associated processes allow for a significant increase in test coverage by mapping test cases into test suites, a reduction of test effort and use of testing resources, and an improvement in software quality. We not only designed and implemented an automated testing system, but also introduced a software development approach for building such testing systems. We began by analyzing the requirements for an automated testing system and then designed the system to satisfy the elicited requirements. The implementation approach adheres to the *software engineering principle separation of concerns*, which resulted in significant maintainability and reusability benefits. In particular, we implemented the test suites according to a functionality decomposition of the subject system. Our automated testing system has been used extensively and was maintained while the tested subject software evolved and changed.

7.2 Contributions

In our work, we have introduced a software development approach for automated testing tools that leads to automated testing systems that are more maintainable, reliable, and practical. To select an appropriate base tool, we conducted considerable research and analyzed and compared testing capabilities of some popular automation tools and methodologies, including data-driven automation frameworks, table-driven automation frameworks, as well as record and

playback tools. We also studied the characteristics of data-driven automation frameworks and identified features offered by these frameworks that might be helpful for other test engineers in testing their software products.

To achieve maintainability, we decomposed the system into various reusable components. We also developed a test suite management solution to improve the usability of the system. Test coverage has been greatly improved as each test case is mapped to specific test functions and test suites. Over 40% of the total test cases were automated and the regression and longevity testing time was reduced by approximately 40%.

By implementing the automated system and applying the system to a case study, we have gained significant experience and knowledge regarding improvements to the automation process and resolution of common problems. We also designed the system with adoption and sustainability in mind.

7.3 Future Work

Developing automated testing systems was a new practice for NewHeights Software Corporation. We had little experience in developing test plans for an automated testing system. These test plans were derived from existing manual test plans and developed incrementally. For

both manual and automated testing, it is difficult to characterize the exact test coverage. With respect to test coverage, naturally there is overlap between manual and automated testing. Thus, it is important that the automation test plans be developed before implementing the automated testing processes for a particular subject system. Well-designed and well-developed test plans can guide and drive the requirements analysis of the automated testing processes.

Our automated testing system covers most of the regression test cases for the tested commercial product. In the future, we could also implement longevity test cases and reduce the testing effort for over-night testing.

There is plenty of room for improvement with respect to the automated testing system and the testing processes it supports. For example, we could further subdivide the reusable components of the automated testing tool as well as the regression test suites to facilitate more fine-grained reusability. We could also investigate how the automated testing system can detect and verify SIP (Session Initiation Protocol) addresses for validating the audio functions of the product.

References

[1] AutomatedQA: TestComplete, 2008.
http://www.automatedqa.com/products/testcomplete/

[2] AutomatedQA: TestComplete—Key Features, 2007.
http://www.automatedqa.com/products/testcomplete/tc_features.asp

[3] AutomatedQA: TestComplete—Functional (GUI) Testing, 2007.
http://www.automatedqa.com/products/testcomplete/tc_functional_testing.asp

[4] Andreas Borchert: Software Quality, 2002.
http://www.cs.rit.edu/~afb/20012/cs1/slides/quality.html

[5] W.J. Brown, R.C. Malveau, H.W. McCormick III, T.J. Mowbray*: Anti-Patterns: Refactoring Software, Architectures, and Projects in Crisis*, John Wiley & Sons, 1998.

[6] W.J. Brown, H.W. McCormick III, S.W. Thomas: *Anti-Patterns in Project Management*, John Wiley & Sons, 2000.

[7] Edsger Dijkstra: *A Discipline of Programming*, Prentice Hall, 1976.

[8] Elfriede Dustin: Lessons in Test Automation: A Manager's Guide to Avoiding Pitfalls when Automating Testing, May 2001.
http://www.awprofessional.com/articles/article.asp?p=21467&rl=1

[9] Simson Garfinkel. History's Worst Software Bugs, August 2005.
http://www.wired.com/software/coolapps/news/2005/11/69355

[10] Anne Geraci (ed.): *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*, The Institute of Electrical and Electronics Engineers Inc., New York, NY, 1991.

[11] Dorothy Graham, Mark Fewster: *Software Test Automation: Effective Use of Test Execution Tools*, ACM Press, 1999.

[12] Linley Erin Hall. Greedy to Get the Bugs Out, 2007.

http://researchmag.asu.edu/stories/bugsout.html

[13] Fred Hapgood: The Importance of Automated Software Testing, *CIO Magazine,* August 2001.
http://www.cio.com/article/print/30432

[14] Linda Hayes: Automated Testing Handbook, Software Testing Institute, 2nd edition, March 2004.

[15] Rick Hower: Will Automated Testing Tools Make Testing Easier? 2008.
http://www.softwareqatest.com/qat_lfaq1.html#LFAQ1_8

[16] Rick Hower: What Kinds of Testing Should Be Considered? 2008.
http://www.softwareqatest.com/qatfaq1.html#FAQ1_10

[17] IBM Corporation: Rational: The Key to Successful Automated Testing: Planning, 2003.
http://www3.software.ibm.com/ibmdl/pub/software/rational/web/whitepapers/2003/planning.pdf

[18] Herbert Isenberg: The Practical Organization of Automated Software Testing, 2007.
http://www.automated-testing.com/PATfinal.htm

[19] Cem Kaner, James Bach, Bret Pettichord: *Lessons Learned in Software Testing,* John Wiley & Sons, 2002.

[20] Cem Kaner: Avoiding Shelfware: A Manager's View of Automated GUI Testing, *Software Testing Analysis and Review Conference (STAR) East*, Orlando, Florida, May 1998.

[21] Cem Kaner: Improving the Maintainability of Automated Test Suites, *Software Quality Assurance,* Vol. 4, No. 4, 1997.

[22] Cem Kaner: Pitfalls and Strategies in Automated Testing, *IEEE Computer*, Vol. 30, No. 4, pp. 114-116, April 1997.

[23] Brian Kauffman, Tom Bettge, Lawrence Buja, Tony Craig, Cecelia DeLuca, Brian Eaton, Matthew Hecht, Erik Kluzek, Jim Rosinski, Mariana Vertenstein: Community Climate System Model: Software Developer's Guide, Subsection 3.1, June 2001.

[24] Michael Kelly, IBM Corporation: Choosing a Test Automation Framework, November 2003.

http://www.ibm.com/developerworks/rational/library/591.html

[25] Atif Memon, Martha Pollack, Mary Lou Soffa: Hierarchical GUI Test Case Generation Using Automated Planning and Strategies in Automated Testing, *IEEE Transactions on Software Engineering*, Vol. 27, No. 2, pp. 144-155, February 2001.

[26] Bret Pettichord: Seven Steps to Test Automation Success. STAR West, San Jose, November 1999.
http://www.io.com/~wazmo/papers/seven_steps.html

[27] Bret Pettichord: Success in Automation, June 2001.
http://www.io.com/~wazmo/succpap.htm

[28] Mauro Pezzè, Michal Young: *Software Testing and Analysis: Process, Principles and Techniques*, John Wiley & Sons, 2007.

[29] Redstone Software: Driving Success Through Automation, 2008.
http://www.redstonesoftware.com/solutions/data_driven

[30] Ian Sommerville: A Software Engineering, Pearson Education, 2006.

[31] Wikipedia: Software Bug.
http://en.wikipedia.org/wiki/Software_bug

[32] Wikipedia: Test Automation, 2008.
http://en.wikipedia.org/wiki/Test_automation

[33] Test Automation Framework, 2008.
http://en.wikipedia.org/wiki/Test_automation_framework

Appendix A: Source Code for Testing Telephony

```
//USEUNIT Tools
//USEUNIT StartupShutdown
//USEUNIT BellDebugger1
//USEUNIT BellDebugger2
//USEUNIT CallUtilities
//USEUNIT ShutterUtilities
//USEUNIT ContactShutter

var sPrefix = "ImagesUsedToCompare\\Telephony\\";
var sPrefixNew = "images\\";
var sPostfix = ".bmp";
function Main()
{
	TestIncomingCall();
	TestOutgoingCalls();
	TestTransferCall();
}

function TestIncomingCall()
{
	Startup(1);
	BellDebugger1Startup();
	BellDebugger2Startup();
	UndockShutter("Contact");
	CreateContactWithPhoneOnly( sSampleContactName, sCompanyName, "P1",
	   sBellDebugger1Address );
	BellDebugger1MakeCall(sTestedProductAddress);
	p = GetP();
	w = p["WaitChild"]( "TelephonyPopupWindow", 1000 );

	var i = 0;
	while ( i < 60 && !w["Exists"])
	{
	    Sys["Delay"](1000);    // wait a second then try again
	    p = GetP();
	    w = p["WaitChild"]( "TelephonyPopupWindow", 1000 );
```

```
    i = i+1;
}
w = GetW("TelephonyPopupWindow");       //changed to 2 and back
Verify( w["Exists"], "Incoming call popped up an incoming call window" );
w["Activate"]();
var expectedCaller = sSampleContactName;
TestMsg(expectedCaller);
var actualCaller = w["nameLabel"]["Text"];
TestMsg(actualCaller);
Verify(actualCaller.indexOf(expectedCaller)>-1, "Caller name are shown on incoming
   popup correctly. Expected: " + expectedCaller + ". Actual: " + actualCaller);

var expectedCallerId = "(sip:"+ sBellDebugger1Address +") is calling on line "+
   sTestedProductAddress;
TestMsg(expectedCallerId);
var actualCallerId = w["numberLabel"]["Text"] + " " + w["lineLabel"]["Text"];
TestMsg(actualCallerId);
Verify(actualCallerId.indexOf(expectedCallerId)>-1, "Caller id are shown on incoming
   popup correctly. Expected: " + expectedCallerId + ". Actual: " + actualCallerId);

AnswerCall();
Sys.Delay(2000);
p = Sys["Process"]("DA"); //added
w = p["WaitChild"]("TelephonyPopupWindow", 1000);    //changed to 2 and back
Verify( !w["Exists"], "incoming popup disappears after call is answered (as expected)" );

w = GetW("CommunicationWindow");
Verify( w["Exists"], "After answers the call, comm window shows up." );
VerifyCommunicationStatus(true, "Talking");
HangupCall();
Sys.Delay(2000);
w = p["WaitChild"]("CommunicationWindow", 1000);
Verify( !w["Exists"], "Comm window disappears after call is hung up." );
w = p["MainWindow"];
w["Activate"]();
var filename = sPrefix + "Icon_Idle" + sPostfix;
var newfilename = sPrefix + "New_Icon_Idle" + sPostfix;
SaveImage( w["iconPictureBox"], newfilename);
Verify(CompareImage(newfilename, filename), "Idle Icon is correct.");

p = Sys["Process"]("DA");
```

```
w = p["KMWindow"];
Verify( w["Exists"], "KM window is open after call connected." );
w["okButton"]["Click"](38, 12);

BellDebugger2MakeCall(sTestedProductAddress,1);
p = GetP();
w = p["WaitChild"]( "TelephonyPopupWindow", 1000 );

var i = 0;
while ( i < 60 && !w["Exists"])
{
    Sys["Delay"](1000);    // wait a second then try again
    p = GetP();
    w = p["WaitChild"]( "TelephonyPopupWindow", 1000 );
    i = i+1;
}
w = GetW("TelephonyPopupWindow");        //change to 2 and back
Verify( w["Exists"], "Incoming call popped up an incoming call window" );
w["Activate"]();
var expectedCaller = sBellDebuggerCallerID; //should have a new var for this
TestMsg(expectedCaller);
var actualCaller = w["nameLabel"]["Text"];
TestMsg(actualCaller);
Verify(actualCaller.indexOf(expectedCaller)>-1, "Caller name are shown on incoming
   popup correctly. Expected: " + expectedCaller + ". Actual: " + actualCaller);

var expectedCallerId = "(sip:"+ sBellDebugger2Address +") is calling on line "+
   sTestedProductAddress;
TestMsg(expectedCallerId);
var actualCallerId = w["numberLabel"]["Text"] + " " + w["lineLabel"]["Text"];
TestMsg(actualCallerId);
Verify(actualCallerId.indexOf(expectedCallerId)>-1, "Caller id are shown on incoming
   popup correctly. Expected: " + expectedCallerId + ". Actual: " + actualCallerId);

Sys.Delay(2000);
AnswerCall();
Sys.Delay(2000);
p = Sys["Process"]("DA");
w = p["WaitChild"]("TelephonyPopupWindow", 1000);
Verify(!w["Exists"], "Incoming popup disappears after BellDebugger2 hangs up.");
```

```
HangupCall();
Sys.Delay(2000);

BellDebugger2MakeCall(sTestedProductAddress,1);
p = GetP();
w = p["WaitChild"]( "TelephonyPopupWindow", 1000 );

var i = 0;
while ( i < 60 && !w["Exists"])
{
    Sys["Delay"](1000);    // wait a second then try again
    p = GetP();
    w = p["WaitChild"]( "TelephonyPopupWindow", 1000 );
    i = i+1;
}
w = GetW("TelephonyPopupWindow");
Verify( w["Exists"], "Incoming call popped up an incoming call window" );

BellDebugger2Hangup();
Sys.Delay(2000);
w = p["WaitChild"]("TelephonyPopupWindow", 1000);
Verify(!w["Exists"], "Incoming popup disappears after BellDebugger2 hangs up.");

BellDebugger2MakeCall(sTestedProductAddress,1);
p = GetP();
w = p["WaitChild"]( "TelephonyPopupWindow", 1000 );
Verify( w["Exists"], "Incoming call popped up an incoming call window" );
w["Activate"]();
w["closeButton"]["Click"](5, 9);
w = p["WaitChild"]("TelephonyPopupWindow", 1000);
Verify(!w["Visible"], "Incoming popup disappears after it was closed.");
UndockShutter("Communications");
w = GetW("CommunicationsShutter");
w["Activate"]();
var filename = sPrefix + "Icon_Ringing" + sPostfix;
var newfilename = sPrefix + "New_Icon_Ringing" + sPostfix;
SaveImage( w["iconPictureBox"], newfilename);
Verify(CompareImage(newfilename, filename), "Icon ringing is correct.");
Sys.Delay(1000);

BellDebugger2Hangup();
```

```
	DeleteFirstContact("phone");
	DockShutter("Contact");
	DockShutter("Communications");
	Shutdown();
	BellDebugger1Shutdown();
	BellDebugger2Shutdown();
}
function TestOutgoingCalls()
{
	InitializeLocalVariables(); //not needed if running Startup()
	Startup(1);
	BellDebugger1Startup();
	BellDebugger2Startup();
	UndockShutter( "Contact" );
	UndockShutter( "History" );
	CreateContactWithPhoneOnly( sSampleContactName, sCompanyName, "phone",
	  sBellDebugger1Address );
	DeleteCallLogEntries();

	CallByQuickConnectKeyboard( sBellDebugger1Address, 9);
	CallByQuickConnectDropDown();
	CallByContextMenu();
	CallByDoubleClickingContact();
	CallByDraggingToQuickConnectBox();
	CallByDraggingToCommShutter();
	CallByFavorite();
	CallByCallLog();
	CallByRightClickOnHistoryShutter();

	MakeCall(sInvalidPhoneNumber, 9);
	VerifyCommunicationStatus(false, "Invalid Number");
	HangupCall();
	var p = GetP();
	var w = p["WaitChild"]("CommunicationWindow", 1000);
	Verify( !w["Exists"], "Comm window disappears after call is hung up." );
	w = p["MainWindow"];
	w["Activate"]();
	var filename2 = sPrefix + "Icon_Idle" + sPostfix;
	var newfilename2 = sPrefix + "New_Icon_Idle2" + sPostfix;
	SaveImage( w["iconPictureBox"], newfilename2);
	Verify(CompareImage(newfilename2, filename2), "Idle Icon is correct.");
```

```
BellDebugger2MakeCall(sBellDebugger1Address, 1);
BellDebugger1AnswerCall();                              //note: win num
Sys.Delay(2000);
BellDebugger2MakeCall(sBellDebugger1Address, 1);
BellDebugger1AnswerCall();
Sys.Delay(2000);
BellDebugger2MakeCall(sBellDebugger1Address, 1);
BellDebugger1AnswerCall();
Sys.Delay(2000);

MakeCall(sBellDebugger2Address, 9);
VerifyCommunicationStatus(false, "Ringing");
HangupCall();
MakeCall(sBellDebugger2Address, 9);
VerifyCommunicationStatus(false, "Ringing");
HangupCall();
MakeCall(sBellDebugger2Address, 9);
VerifyCommunicationStatus(false, "Ringing");
HangupCall();
MakeCall(sBellDebugger1Address, 9);
VerifyCommunicationStatus(false, "Ringing");
HangupCall();
HangupCallsToTerminateBusyState("BellPCMDebug1");

MakeCall(sSilverCityAddress, 9);
VerifyCommunicationStatus(false, "Dialing");
Sys.Delay(8000);    //needed longer delay
VerifyCommunicationStatus(true, "Talking");
HangupCall();
p = GetP();
w = p["WaitChild"]("CommunicationWindow", 1000);
Verify( !w["Exists"], "Comm window disappears after call is hung up." );
w = p["MainWindow"];
w["Activate"]();
var filename3 = sPrefix + "Icon_Idle" + sPostfix;
var newfilename3 = sPrefix + "New_Icon_Idle3" + sPostfix;
SaveImage( w["iconPictureBox"], newfilename3);
Verify(CompareImage(newfilename3, filename3), "Idle Icon is correct.");

MakeCall(sInvalidPhoneNumber + "@" + sPSTNGateway, 9);
```

```
	BellDebugger2MakeCall(sTestedProductAddress, 1);
	var expectedBellDebugger2Status = "Ringing";
	var actualBellDebugger2Status =
	  BellDebugger2GetCommunicationWindowStatusLabel1();
	Verify(actualBellDebugger2Status.indexOf(expectedBellDebugger2Status) > -1, "Only 1
	  line is busy for PCM, call rings.");
	BellDebugger2Hangup();
	MakeCall(sInvalidPhoneNumber + "@" + sPSTNGateway, 9);
	MakeCall(sInvalidPhoneNumber + "@" + sPSTNGateway, 9);
	MakeCall(sInvalidPhoneNumber + "@" + sPSTNGateway, 9);
	BellDebugger2MakeCall(sTestedProductAddress, 1);
	expectedBellDebugger2Status = "Busy";
	Sys.Delay(4000);
	actualBellDebugger2Status = BellDebugger2GetCommunicationWindowStatusLabel1();
	TestMsg(expectedBellDebugger2Status);
	TestMsg(actualBellDebugger2Status);
	Verify(actualBellDebugger2Status.indexOf(expectedBellDebugger2Status) > -1, "PCM is
	  all busy, call goes to Voicemail.");
	BellDebugger2Hangup();
	HangupCallsToTerminateBusyState("DA");

	BellDebugger2MakeCall(sTestedProductAddress, 1);
	AnswerCall();
	VerifyCommunicationStatus(true, "Talking");
	HangupCall();

	DockShutter( "Contact" );
	DockShutter( "History" );    //not working for me because mine is _2
	BellDebugger1Shutdown();
	BellDebugger2Shutdown();
	Shutdown();
}

function TestTransferCall()
{
	InitializeLocalVariables(); //not needed if Startup is run
	Startup(1);
	UndockShutter("Contact");
	UndockShutter("History");
	BellDebugger2Startup();
	BellDebugger1Startup();
```

```
TestMsg("Verify a call transfer works correctly (on outgoing call)");
MakeCall( sBellDebugger2Address, 9);
Sys.Delay(500);
BellDebugger2AnswerCall();
Sys.Delay(500);
TransferCall( sBellDebugger1Address );
Sys.Delay(1000);
BellDebugger1AnswerCall();
Sys.Delay(1000);
Verify( BellDebugger2IsConnected(), "BellDebugger2's status is: CONNECTED" );
BellDebugger2Hangup();
Sys.Delay(500);

TestMsg("Transfer an outgoing SIP call to a PSTN number, transfer works.");
MakeCall(sBellDebugger2Address, 9);
BellDebugger2AnswerCall();
TransferCall(sSilverCityAddress);

Sys.Delay(8000);
w = BellDebugger2GetW("CommunicationWindow");

var i = 0;
////////////////// dialing delay issue ////////////////////////
while ( i < 60 && BellDebugger2GetCommunicationWindowStatusLabel().indexOf("Held
   By") > -1)
{
    Sys["Delay"](1000);    // wait a second then try again
    i = i+1;
}
p = Sys["Process"]("DA");        //added
w = p["WaitChild"]("CommunicationWindow", 2000);
Verify( !w["Exists"], "Comm window disappears after call is transfered (as expected)" );
Verify( BellDebugger2IsConnected(), "BellDebugger2's status is: CONNECTED, with
   external number" );
BellDebugger2Hangup();

TestMsg("Transfer an outgoing SIP call to an invalid SIP URI, transfer doesn't work.");
MakeCall(sBellDebugger2Address, 9);
BellDebugger2AnswerCall();
TransferCall(sInvalidPhoneNumber);
```

```
VerifyCallStateLabel("Invalid Number");
HangupCall(); //Hang Up Consult
VerifyCallStateLabel("Talking");
HangupCall();

MakeCall(sBellDebugger2Address, 9);
VerifyCommunicationStatus( false, "Ringing" );
HangupCall();
p = Sys["Process"]("DA");       //added
w = p["WaitChild"]("CommunicationWindow", 1000);
Verify( !w["Exists"], "Comm window disappears after call is hung up." );
MakeCall(sSilverCityAddress, 9);
Sys.Delay(12000);    //increased delay for PSTN call to connect
TransferCall(sBellDebugger2Address);
Sys.Delay(2000);
//TestedPCM in Transferring state, call ringing for Debug2
VerifyCallStateLabel("Transferring");
w = GetW("CommunicationWindow");
Verify( w["ccKey1"]["Enabled"] == false, "Hold button is disabled");
Verify( w["ccKey2"]["Enabled"] == false, "Hang Up button is disabled");
Verify( w["ccKey3"]["Enabled"] == true, "Release Me button is enabled");
Verify( w["ccKey4"]["Enabled"] == false, "Join All button is disabled");
Verify( w["ccKey5"]["Enabled"] == false, "Switch button is disabled");
BellDebugger2AnswerCall();
Sys.Delay(2000);
p = Sys["Process"]("DA");    //added
w = p["WaitChild"]("CommunicationWindow", 1000);
Verify( !w["Exists"], "Comm window disappears after call is transfered (as expected)" );
Verify( BellDebugger2IsConnected(), "BellDebugger2's status is: CONNECTED, with
   external number" );
BellDebugger2Hangup();

MakeCall(sSilverCityAddress, 9);
Sys.Delay(10000);    //increased delay
TransferCall(sInvalidPhoneNumber);
Sys["Delay"](3000);    // wait a second then try again
VerifyCallStateLabel("Invalid Number" );
//UnHoldCall("Invalid Number");      //different in PCM
HangupCall(); //Hang Up Consult
```

```
VerifyCallStateLabel("Talking");
HangupCall();

MakeCall(sSilverCityAddress, 9);
VerifyCommunicationStatus( false, "Dialing" );
Sys.Delay(8000);
HangupCall();
p = Sys["Process"]("DA");    //added
w = p["WaitChild"]("CommunicationWindow", 1000);
Verify( !w["Exists"], "Comm window disappears after call is transfered (as expected)" );

BellDebugger2MakeCall( sTestedProductAddress);
Sys.Delay(500);
UndockShutter("Communications");
w = GetW("CommunicationsShutter");
//w["partyRichLabel"]["Click"](80, 11); //dif label in PCM
w["lineLabel"]["Click"](36, 7);
VerifyCallStateLabel( "Ringing" );

AnswerCall();
Sys.Delay(500);
TransferCall( sBellDebugger1Address );
Sys.Delay(1000);
VerifyCallStateLabel("Transferring");
BellDebugger1AnswerCall();
Sys.Delay(1000);
Verify( BellDebugger2IsConnected(), "BellDebugger2's status is: CONNECTED" );
Verify( BellDebugger1IsConnected(), "BellDebugger1's status is: CONNECTED" );
Sys.Delay(1000);
p = Sys["Process"]("DA");
w = p["WaitChild"]("CommunicationWindow", 1000);
Verify( !w["Exists"], "PCM's Comm window disappeared after call is transfered
   (expected)." );
BellDebugger2Hangup();
Sys.Delay(500);

BellDebugger2MakeCall( sTestedProductAddress);
AnswerCall();
TransferCall(sSilverCityAddress);
Sys.Delay(3000);
VerifyCallStateLabel("Transferring");
```

```
Sys.Delay(8000);
p = GetW();
w = p["WaitChild"]("CommunicationWindow", 1000);
Verify( !w["Exists"], "Comm window disappears after call is transfered (as expected)" );
Sys.Delay(4000);
Verify( BellDebugger2IsConnected(), "BellDebugger2's status is: CONNECTED, with external number" );
BellDebugger2Hangup();

BellDebugger2MakeCall( sTestedProductAddress);
AnswerCall();
HoldCall();
TransferCall( sBellDebugger1Address );
Sys.Delay(1000);
BellDebugger1AnswerCall();
Sys.Delay(1000);
Verify( BellDebugger2IsConnected(), "BellDebugger2's status is: CONNECTED" );
Verify( BellDebugger1IsConnected(), "BellDebugger1's status is: CONNECTED" );
Sys.Delay(1000);
p = Sys["Process"]("DA");
w = p["WaitChild"]("CommunicationWindow", 1000);
Verify( !w["Exists"], "PCM's Comm window disappeared after call is transfered (expected)." );
BellDebugger2Hangup();

MakeCall( sBellDebugger2Address, 9);
BellDebugger2AnswerCall();
HoldCall();
TransferCall( sBellDebugger1Address );
Sys.Delay(1000);
BellDebugger1AnswerCall();
Sys.Delay(1000);
Verify( BellDebugger2IsConnected(), "BellDebugger2's status is: CONNECTED" );
Verify( BellDebugger1IsConnected(), "BellDebugger1's status is: CONNECTED" );
Sys.Delay(1000);
p = Sys["Process"]("DA");
w = p["WaitChild"]("CommunicationWindow", 1000);
Verify( !w["Exists"], "PCM's Comm window disappeared after call is transfered (expected)." );
BellDebugger2Hangup();
```

```
    BellDebugger1Shutdown();
    BellDebugger2Shutdown();
    Shutdown();
}
```

Appendix B: Source Code for Testing Communications Window

```
//USEUNIT Tools
//USEUNIT StartupShutdown
//USEUNIT ShutterUtilities
//USEUNIT BellDebugger1
//USEUNIT BellDebugger2
//USEUNIT ContactShutter

var sPrefix = "ImagesUsedToCompare\\CommunicationWindow\\";
var sPrefixNew = "images\\";
var sPostfix = ".bmp";

function Main()
{
	TestCommunicationWindowUI();
	TestCommunicationWindowParticipantsList();
	TestCommunicationDifferentStates();
}

function TestCommunicationWindowUI()
{
	InitializeLocalVariables();
	Startup(1);
	BellDebugger1Startup();
	BellDebugger2Startup();
	UndockShutter("Contact");

	MakeCallByDoubleClickingContact(sBellDebugger2Address);
	BellDebugger2AnswerCall();
	Sys.Delay(1000);
	var w;

	//MainWindow's Left, Top, Width and Height
	var ml, mt, mw, mh;
	w = GetW("CommunicationWindow");
	w["Activate"]();
	w["textLabel"]["Click"](48, 3);
```

```
GetHelpWindow();

ResizeAndMoveCommunicationWindow();
w = GetW("CommunicationWindow");
ml = w["Left"];
mt = w["Top"];
mw = w["Width"];
mh = w["Height"];
Log["Message"]("Communication Windows Left:" + ml + ", Top:" + mt + ", Width:" + mw + " and Height:" + mh);
MinimizeCommWindow();
MaximizeCommWindow();

w = GetW("CommunicationWindow");
ml2 = w["Left"];
mt2 = w["Top"];
mw2 = w["Width"];
mh2 = w["Height"];

Verify((ml == ml2)&&(mt == mt2)&&(mw == mw2)&&(mh == mh2), "Comm window size is persistent.");

CloseCommWindow();
ReopenCommWindow();

w = GetW("CommunicationWindow");
ml2 = w["Left"];
mt2 = w["Top"];
mw2 = w["Width"];
mh2 = w["Height"];

Verify((ml == ml2)&&(mt == mt2)&&(mw == mw2)&&(mh == mh2), "Comm window size is persistent.");
HangupCall();
Sys.Delay(3000);

MakeCallByDoubleClickingContact(sBellDebugger2Address);
Sys.Delay(1000);
BellDebugger2AnswerCall();
Sys.Delay(1000);
```

```
w = GetW("CommunicationWindow");
ml2 = w["Left"];
mt2 = w["Top"];
mw2 = w["Width"];
mh2 = w["Height"];
Verify((ml == ml2)&&(mt == mt2)&&(mw == mw2)&&(mh == mh2), "Comm window size is persistent for the second call.");
HangupCall();
Shutdown();
Startup();
MakeCallByDoubleClickingContact(sBellDebugger2Address);
Sys.Delay(1000);
BellDebugger2AnswerCall();
Sys.Delay(1000);
w = GetW("CommunicationWindow");
ml2 = w["Left"];
mt2 = w["Top"];
mw2 = w["Width"];
mh2 = w["Height"];

Verify((ml == ml2)&&(mt == mt2)&&(mw == mw2)&&(mh == mh2), "Comm window size is persistent in height for the second call after restarting.");
HangupCall();

MakeCallByDoubleClickingContact(sBellDebugger2Address);
VerifyCommunicationStatus(false, "Ringing");
BellDebugger2AnswerCall();
VerifyCommunicationStatus(true, "Talking");

w = GetW("CommunicationWindow");
var expectedCalleeInfo = (sBellDebuggerCallerID + " (sip:" + sBellDebugger2Address + ")");
var actualCalleeInfo = w["infoLabel"]["Text"];
Verify(actualCalleeInfo.indexOf(expectedCalleeInfo) > -1, "Callee info Unknown shown is correct: " + expectedCalleeInfo + " as expected");
HangupCall();

p = Sys["Process"]("DA");
w = p["WaitChild"]("CommunicationWindow", 1000);
Verify( !w["Exists"], "Comm window disappears after call is transfered (as expected)" );
```

```
	MakeCallByDoubleClickingContact(sBellDebugger1Address);
	VerifyCommunicationStatus(false, "Ringing");
	w = GetW("CommunicationWindow");
	var expectedCalleeInfo = (sBellDebuggerCallerID + " (sip:" + sBellDebugger1Address +
")");
	var actualCalleeInfo = w["infoLabel"]["Text"];
	Verify(actualCalleeInfo.indexOf(expectedCalleeInfo) > -1, "Callee info Unknown shown is
correct: " + expectedCalleeInfo + " as expected");
	HangupCall();

	CreateContactWithPhoneOnly("With ContactMatch", "NH", "phone",
sBellDebugger1UserName);
	w = GetW("ContactShutter");
	w["contactLabel"]["DblClick"](73, 5);
	VerifyCommunicationStatus(false, "Ringing");
	w = GetW("CommunicationWindow");
	var expectedCalleeInfo = ("With ContactMatch (sip:" + sBellDebugger1Address + ")");
	var actualCalleeInfo = w["infoLabel"]["Text"];
	Verify(actualCalleeInfo.indexOf(expectedCalleeInfo) > -1, "Callee info Unknown shown is
correct: " + expectedCalleeInfo + " as expected");
	HangupCall();

	DeleteFirstContact("phone");
	MakeCallByDoubleClickingContact(sPhoneNumberSilverCity);
	VerifyCommunicationStatus(false, "Dialing");
	Sys.Delay(7000);
	VerifyCommunicationStatus(true, "Talking");
	HangupCall();

	w = p["WaitChild"]("CommunicationWindow", 1000);
	Verify( !w["Exists"], "Comm window disappears after call is transfered (as expected)" );

	DockShutter("Contact");
	Shutdown();
	BellDebugger1Shutdown();
	BellDebugger2Shutdown();
}

function TestCommunicationWindowParticipantsList()
{
```

```
InitializeLocalVariables();
Startup(1);
BellDebugger1Startup();
UndockShutter("Contact");
CreateContactWithAllData(sSampleContactName, sCompanyName,
                         "P1", sBellDebugger1Address,
                         "E1", sValidEmail,
                         "MSN1", sValidMSN );
var w, p;
p = GetP();
DialFirstContactInContactShutter();
BellDebugger1AnswerCall();

w = GetW("CommunicationWindow");
w["infoLabel"]["ClickR"](53, 8);
Sys["Delay"](500);

Var filename = sPrefix + "ContextMenu_ CommunicationWindow_WithMSN_withEmail"
+ sPostfix;
var newfilename = sPrefix +
"New_ContextMenu_CommunicationWindow_WithMSN_WithEmail" + sPostfix;
SaveContextMenuImage(newfilename);
Verify(CompareImage(newfilename, filename), "Context menu shown in comm window is
   correct.");

w["PopupMenu"]["Click"]("[0]");
p = Sys["Process"]("WINWORD");
w = p["Window"]("OpusApp", "Untitled Message");
w["Activate"]();
Verify(w["Exists"], "Email throught context menu is successful.");
w["Close"]();
w = p["Window"]("#32770", "Microsoft Office Word");
w["Activate"]();
w["Window"]("Button", "&No")["Click"]();

w = GetW("CommunicationWindow");
w["Activate"]();
w["infoLabel"]["ClickR"](151, 8);
w["PopupMenu"]["Click"]("[1]");
p = Sys["Process"]("msnmsgr");
w = p["Window"]("IMWindowClass", "*");
```

```
w["Activate"]();
Verify(w["Exists"], "MSN by context menu is successful.");
w["Close"]();

p = Sys["Process"]("DA");
w = p["CommunicationWindow"];
w["Activate"]();
w["infoLabel"]["ClickR"](173, 11);
w["PopupMenu"]["Click"]("[4]");

w = p["WaitWindow"]("*", "Personal Contact Information", -1, 1000);
Verify( w["Exists"],
     "Open Contact Information Window by using the context menu of the contact in
   Communication window");
w["okButton"]["Click"](50, 11);
w = p["CommunicationWindow"];
w["Activate"]();
w["infoLabel"]["ClickR"](173, 11);
w["PopupMenu"]["Click"]("[2]");
w = p["WaitChild"]("KMWindow", 1000);
Verify( w["Exists"],
     "Open KM Window by using the context menu of the contact in Communication
   window");
w["okButton"]["Click"](50, 11);
HangupCall();

CreateContactWithPhoneOnly("Second Contact", sCompanyName, "Phone",
   sBellDebugger1Address);

w = GetW("ContactShutter");
var expectedFirstContact = sSampleContactName;
var actualFirstContact = w["contactLabel"]["Text"];
Verify(expectedFirstContact.indexOf(actualFirstContact) > -1, "Contact is sorted, first
   contact:" + expectedFirstContact + " as expected");
var expectedSecondContact = "Second Contact";
var actualSecondContact = w["contactLabel_2"]["Text"];
Verify(expectedFirstContact.indexOf(actualFirstContact) > -1, "Contact is sorted, Second
   contact:" + expectedSecondContact + " as expected");

MakeCall(sBellDebugger1Address, 9);
p = Sys["Process"]("DA");
```

```
    VerifyCommunicationStatus(false, "Ringing");
    w = GetW("CommunicationWindow");
    var expectedCalleeInfo = ("(2) Contacts (sip:" + sBellDebugger1Address + ")");
    var actualCalleeInfo = w["infoLabel"]["Text"];
    Verify(actualCalleeInfo.indexOf(expectedCalleeInfo) > -1, "Callee info Unknown shown is
       correct: " + expectedCalleeInfo + " as expected");

    w["infoLabel"]["Click"](151, 8);
    w = p["ContactChooserDialog"];
    w["Activate"]();
    w["itemLabel_2"]["Click"](43, 6);
    w["okButton"]["Click"](24, 7);
    w = GetW("CommunicationWindow");
    var expectedCalleeInfo = ("Second Contact (sip:" + sBellDebugger1Address + ")");
    var actualCalleeInfo = w["infoLabel"]["Text"];
    Verify(actualCalleeInfo.indexOf(expectedCalleeInfo) > -1, "Callee info Unknown shown is
       correct: " + expectedCalleeInfo + " as expected");
    HangupCall();
    DockShutter("Contact");
    BellDebugger1Shutdown();
    Shutdown();
}

function TestCommunicationDifferentStates()
{
        Log.Message("Test different status of communication window.");
        InitializeLocalVariables();
        Startup();
        BellDebugger1Startup();
        BellDebugger2Startup();

        MakeCall(sBellDebugger2Address, 8);
        Sys.Delay(1000);
        TestMsg("Test ringing status.");
        VerifyCommunicationStatus(false, "Ringing");
        BellDebugger2AnswerCall();
        Sys.Delay(1000);

        TestMsg("Test talking status.");
        VerifyCallStateLabel("Talking");
```

```
VerifyCommunicationStatus(true, "Talking");
HangupCall();

BellDebugger2MakeCall(sTestedProductAddress, 1);
AnswerCall();
VerifyCommunicationStatus(true, "Talking");
BellDebugger2Hangup();

MakeCall(sBellDebugger2Address, 8);
BellDebugger2AnswerCall();

HoldCall();
UnHoldCall();
BellDebugger2HoldAndUnholdCall();
Sys.Delay(1000);
VerifyCallStateLabel("Held By");
var w = GetW("CommunicationWindow");
Verify( w["ccKey3"]["Enabled"] == false, "Transfer button is disabled");

BellDebugger2HoldAndUnholdCall();
Sys.Delay(1000);
VerifyCallStateLabel("Talking");
BellDebugger2HoldAndUnholdCall();
Sys.Delay(1000);
HoldCall();
Sys.Delay(1000);
VerifyCallStateLabel("On Hold");
var w = GetW("CommunicationWindow");
Verify( w["ccKey3"]["Enabled"] == false, "Transfer button is disabled");
HangupCall();

TestMsg("Test Busy status.");
BellDebugger2MakeCall("00000", 1);
Sys.Delay(2000);
BellDebugger2MakeCall("23456", 1);
Sys.Delay(2000);
BellDebugger2MakeCall("12345", 1);
Sys.Delay(2000);
BellDebugger2MakeCall("12345", 1);
Sys.Delay(1000);
MakeCall(sBellDebugger2Address, 8);
```

```
        Sys.Delay(1000);
        VerifyCommunicationStatus(false, "Busy");
        HangupCall();

        BellDebugger2HangupDifferentInvalidCall();

        TestMsg("Test DND status.");
        SetAndUnsetBellDebugger2ToDND();
        MakeCall(sBellDebugger2Address, 8);
        Sys.Delay(1000);
        VerifyCommunicationStatus(false, "Unavailable");
        HangupCall();

        TestMsg("Test Auto Answer status.");
        SetAndUnsetBellDebugger2ToDND();
        SetAndUnsetBellDebugger2ToAA();
        MakeCall(sBellDebugger2Address, 8);
        Sys.Delay(2000);
        VerifyCommunicationStatus(true, "Talking");
        HangupCall();
        SetAndUnsetBellDebugger2ToAA();

        Shutdown();
        BellDebugger1Shutdown();
        BellDebugger2Shutdown();
}
```

Appendix C: Source Code for Testing MSN Integration

```
//USEUNIT Tools
//USEUNIT StartupShutdown
//USEUNIT ShutterUtilities
//USEUNIT ContactShutter
//USEUNIT BellDebugger1
//USEUNIT CallUtilities

function Main()
{
      TestMSNIcon();
      TestMSNLaunch();
}

function TestMSNIcon()
{
     var sPrefix = "ImagesUsedToCompare\\MSN\\";
     var sPostfix = ".bmp";
      Startup(1);
      UndockShutter("Contact");

      var p = Sys["Process"]("DA");
      CreateContactWithMSNOnly("Contact with MSN Only", sCompanyName, "MSN Address
         1", sValidMSN);
      Sys.Delay(2000);
      w = GetW("ContactShutter");
      w["Activate"]();
      oIcon = w["WaitChild"]("icon0", 1000);
     Verify( oIcon["Exists"], "Contact with a valid MSN address correctly shows MSN icon" );
      DeleteFirstContact( "msn" );

      CreateContactWithMSNOnly("Contact with MSN Only", sCompanyName, "MSN address
         1", "invalidMSN@newheights", true);
      var p = Sys["Process"]("DA");
      w = p["WaitChild"]("MessageBoxForm", 1000);
      Verify(w["Exists"], "Warning message about non existing MSN address appears.");
      w["textLabel"]["Drag"](124, 2, 341, 3);
```

```
    w["button0"]["Click"](28, 9);
    w = GetW("ContactInformationWindow");
    w["Activate"]();
    w["okButton"]["Click"](52, 16);
    Sys.Delay(2000);
    w = GetW("ContactShutter");
    w["Activate"]();
    w["contactLabel"]["Click"](37, 9);
    filename = sPrefix + "FocusedGroup1" + sPostfix;
    var newfilename = sPrefix + "New_InvalidMSN_Without_MSN" + sPostfix;
    SaveImage(w["contactLabel"], newfilename);
    Verify(CompareImage(newfilename, filename), "Focused group 1 icon is correct.");

    oIcon = w["WaitChild"]("icon0", 1000);
    Verify( oIcon["Exists"], "Contact with and invalid MSN address does not show MSN
      icon" );
    DeleteFirstContact( "none" );

    CreateContactWithEmailOnly("Contact with Email Only", sCompanyName, "Work Email",
      "emailonly@newheights.com" );
    Sys.Delay(2000);
    w = GetW("ContactShutter");
    w["Activate"]();
    oIcon = w["WaitChild"]("icon0", 1000);
    Verify( oIcon["Exists"], "Contact without an MSN entry does not display the MSN icon" );
    DeleteFirstContact( "email" );

    DockShutter("Contact");
    Shutdown();
  }

function TestMSNLaunch()
{
    InitializeLocalVariables();
    Startup(1);
    BellDebugger1Startup();
    UndockShutter("Contact");

    CreateContactWithMSNOnly("Contact with MSN Only", sCompanyName, "MSN address
      1", sValidMSN);
    Sys.Delay(2000);
```

```
w = GetW("ContactShutter");
oIcon = w["WaitChild"]("icon0", 1000);
Verify( oIcon["Exists"], "Contact with a valid MSN address correctly shows MSN icon" );
w["icon0"]["DblClick"](5, 8);
VerifyIMWindowExists();
CloseIMWindow();

w = GetW("ContactShutter");
w["contactLabel"]["ClickR"](38, 9);
w["PopupMenu"]["Click"]("[0]");
VerifyIMWindowExists();
CloseIMWindow();

DeleteFirstContact( "msn" );
CreateContactWithAllData("Contact with All Data", sCompanyName,
                                    "P1", sBellDebugger1Address,
                                    "E1", sValidEmail,
                                    "MSN1", sValidMSN );
BellDebugger1MakeCall( sTestedProductAddress, 1 );
Sys.Delay(2000);

ClickOnIncomingPopup();
VerifyIMWindowExists();
CloseIMWindow();

AnswerCall();
Sys.Delay(1000);
VerifyCommunicationStatus( true, "Talking" );
HangupCall();
Sys.Delay(1000);

BellDebugger1MakeCall( sTestedProductAddress, 1 );
Sys.Delay(1000);
AnswerCall();
Sys.Delay(1000);
VerifyCommunicationStatus( true, "Talking" );
ClickOnCommWindow();
VerifyIMWindowExists();
CloseIMWindow();
```

```
    w = GetW("CommunicationWindow");
    w["infoLabel"]["ClickR"](94, 7);
    w["PopupMenu"]["Click"]("[1]");
    VerifyIMWindowExists();
    CloseIMWindow();

    w = GetW("MainWindow");
    w = GetW("ContactShutter");
    w["contactLabel"]["ClickR"](22, 11);
    w["PopupMenu"]["Click"]("[2]");

    VerifyIMWindowExists();
    CloseIMWindow();

    DeleteFirstContact( "all" );
    HangupCall();
    DockShutter("Contact");
    Shutdown();
    BellDebugger1Shutdown();
}
```