System Testing of Desktop and Web Applications

James M. Slack
slack@mnsu.edu
Information Systems & Technology
Minnesota State University
Mankato, MN 56001, USA

Abstract

We want our students to experience system testing of both desktop and web applications, but the cost of professional system-testing tools is far too high. We evaluate several free tools and find that AutoIt makes an ideal educational system-testing tool. We show several examples of desktop and web testing with AutoIt, starting with simple record/playback and working up to a keyword-based testing framework that stores test data in an Excel spreadsheet.

Keywords: system-testing tools, keyword-based tests, record/playback, AutoIt

1. Introduction

In our software-testing course, we emphasize testing from the quality assurance (QA) perspective in the first half and from the developer perspective in the second. In the second half, students learn about unit testing and write test cases in JUnit (JUnit.org, 2010) and Java to reinforce concepts. This part of the course has worked well for several years.

For the QA half of the course, students learn about system testing and write test cases directly from specifications. An example specification might be that the application is password protected. A system-level test case could try to access a protected area of the application without logging in first.

We needed a system-testing tool to reinforce these concepts on desktop GUI and on web applications. We wanted to use just one system-testing tool that works with both application types, so students spend less time learning the tool and more time learning concepts.

In the Spring 2010 semester, we found that AutoIt (AutoIt, 2010) works well as an educational system-testing tool.

2. Literature Review

System testing evaluates whether the complete system meets its specification by observing the behavior of that system (Pezzè & Young, 2008). System testing involves checking functionality, security, performance, usability, accessibility, among other features. For this paper, we are concerned primarily with functionality system testing: ensuring that the system performs all of its required functions correctly, primarily via the system's user interface.

A particular characteristic of system testing is that it is largely independent of the system's development language (Pezzè & Young, 2008). This means that the tester can use a different programming language and even a different programming paradigm when writing system tests.

Garousi and Mathur state the need for student experience with commercial tools: "*In order to effectively teach software engineering students how to solve real-world problems, the software tools, exercises, projects and assignments chosen by testing educators should be practical and realistic. In the context of software testing education, the above need implies the use of realistic and relevant System Under Test (SUT),* *and making use of realistic commercial testing tools. Otherwise, the skills that students acquire in such courses will not enable them to be ready to test large-scale industrial software systems after graduation." (2010, p.91)*

Garousi & Mathur (2010) found that of seven randomly-selected North America universities, just two use any commercial testing software: the University of Alberta, which uses IBM Rational Functional Tester (IBM, 2010); and Purdue, which uses Telcordia AETG Web Service (Telcordia, 2010). (Both universities also use open-source testing tools.) Of the seven universities in the survey, five use JUnit, usually along with other tools.

Buchmann, Arba, and Mocean (2009) used AutoIt to develop an elegant GUI test case execution program that reads test case information from a text file. For each test case, the program executes a user-defined AutoIt function to manipulate the SUT, and then compares the SUT with expected behavior. The program can check standard Window GUI controls and even images.

3. Evaluation

Evaluation Criteria

We try to give students a QA system-testing experience that is as close to the "real thing" as using JUnit is for unit testing. Ideally, we would use a popular commercial-quality tool such as HP QuickTest Pro (Hewlett-Packard Development Company, 2010) for system testing, but the per-student licensing costs are too high. (We briefly considered licensing commercial software for a lab, but nearly all our students have their own computers and prefer to use them for their assignments.) Therefore, we needed a free, Windows-based tool with these features of commercial-quality tools:

Record/playback: The tool should be able to record keyboard and mouse activity into a script for later playback, so students become familiar with the advantages and disadvantages of this simple technique.

Programmability: The tool should use an easy-to-learn, high-level, interpreted language. This capability allows students to move beyond record/playback, building high-level functions for interacting with the SUT, and to construct their own test frameworks.

Desktop GUI and web application support: The tool should be able to test both major application areas: desktop applications (Windows GUIs) and web-based applications.
External resource access: The tool should be able to access files, databases, spreadsheets, and other resources, so that students can store test data in these places and so they can verify application activity.

Control information: The tool should include the ability to find input and output controls and provide information about them. This capability allows students to write higher-level functions to test the SUT.

Integrated development environment (IDE): The tool should include an easy-to-use environment for building and running tests.

Support: The tool should include complete, well written, and well-organized documentation.

Over the past few semesters, we have tried JUnit, Badboy (Badboy Software, 2010), and Selenium (Selenium Project, 2010) for system testing. In Spring 2010, we decided to examine AutoIt and AutoHotKey (AutoHotKey, 2010). This section compares the relative merits of each of these tools.

JUnit

JUnit was originally designed for unit testing, so it is unsuitable for system testing by itself. However, several third-party utilities add system-testing capabilities to JUnit. For example, we have used HttpUnit (Gold, 2010) and HtmlUnit (Gargoyle Software Inc., 2010) for web testing with JUnit, and Abbot (Wall, 2008) for GUI testing.

We have had some success with these third-party tools, but we have found that both HttpUnit and HtmlUnit execute slowly. Furthermore, neither includes record/playback capabilities. Although Abbot does include record/playback for desktop GUIs, it works only with Java Swing and AWT. These drawbacks motivated us to consider other approaches.

Badboy

Figure 1 shows Badboy, a web-testing tool that includes a script editor and an integrated web browser. Of all the tools mentioned, Badboy is by far the easiest to get started with, because it installs easily and excels at record/playback.

Badboy's integrated help file includes several well-written tutorials.

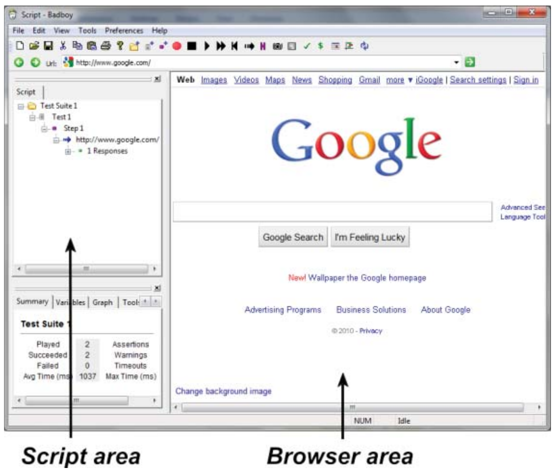


Figure 1: Badboy.

Although Badboy includes load testing, reports, and other valuable features, it has limited programmability and access to external resources, and is useful only for web testing. It cannot test desktop GUI applications, which removes it from further consideration.

Selenium

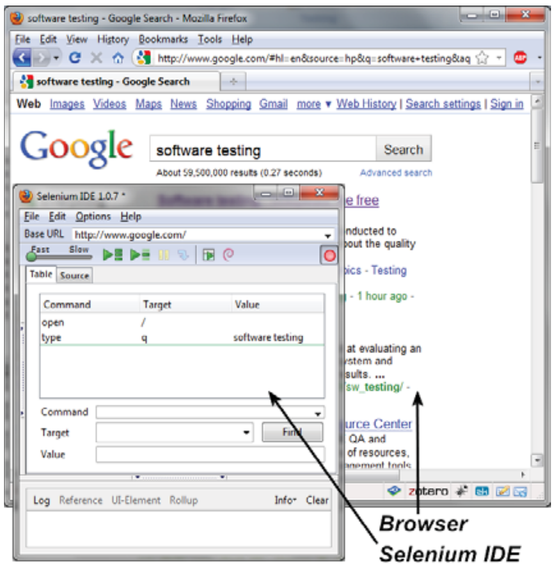


Figure 2: Selenium.

Figure 2 shows Selenium, which is similar to Badboy because it includes a script editor, has very good record/playback support, and only does web testing. In contrast to Badboy, Selenium is a Firefox add-on rather than an integrated application. However, the process of recording and executing scripts is nearly the same as Badboy.

Selenium has a great deal of well-written documentation and an active user community. Selenium can convert its scripts to several different formats, including Java (JUnit), Python, Ruby, C#, Perl, and PHP. This capability makes these scripts easy to customize with higher-level functions and external resources.

Selenium has the same major drawback as Badboy: it works only for web applications. We needed a tool that works with both web and desktop applications.

AutoHotKey and AutoIt

AutoHotKey and AutoIt are each automation utilities for Windows that are very similar to each other. This similarity is not surprising because AutoHotKey started as a fork of AutoIt in 2003 (Wikipedia, 2010).

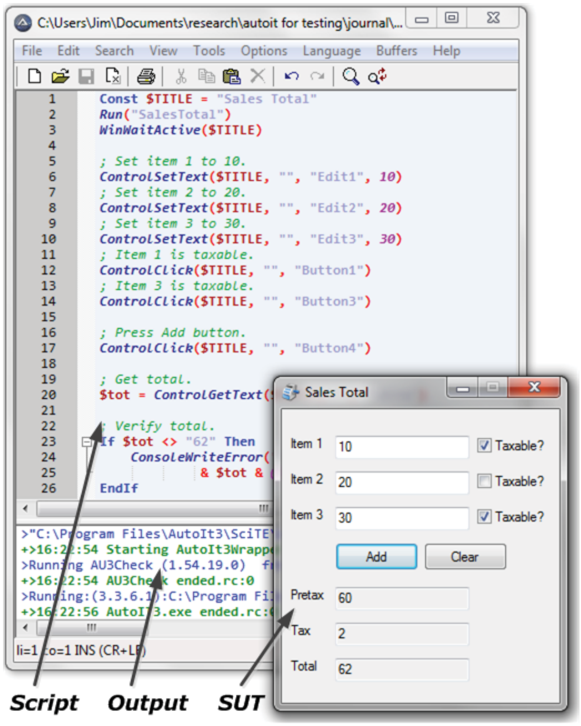
Neither utility was designed specifically for testing, but they can be used that way because each includes a simple scripting language, record/playback capability, the ability to access external resources, and a simple IDE built on the SciTE editor (SciTE, 2010). They can each generate GUI executables, which is convenient for creating desktop SUTs. Each has a well-written help file and an active user community.

Of the two, we have found AutoIt to be generally more robust and better documented. In addition, AutoIt has a much larger standard library that includes functions for accessing and controlling SQLite databases, Excel spreadsheets, and the Internet Explorer browser. AutoHotKey can do all this, too, but requires installing third-party libraries. (Both can access other external resources with ActiveX.)

Finally, we have found that AutoIt's programming language is easier for students to learn, because it is similar to Visual Basic (VB). In contrast, AutoHotKey's programming language is similar to MS-DOS batch language, which most of our students are not familiar with, in spite of using Windows.

We prefer a VB-like language, because HP QuickTest Pro uses VB, and we want students to get a feel for professional testing tools. Figure 3 shows the AutoIt IDE with a test script at the top, results of the test at the bottom, and a simple SUT created with AutoIt's GUI facility.

Table 1 (in the appendix) summarizes the author's subjective evaluation of the testing tools we considered. The rating scale goes from zero (not present) to five (excellent support). AutoIt emerges as the clear winner in this evaluation.


C:\Users\Jim\Documents\research\autoit for testing\journal\...
File Edit Search View Tools Options Language Buffers Help
Sales Total
Add
Clear
Script
Output
SUT

```
Const $TITLE = "Sales Total"
Run("SalesTotal")
WinWaitActive($TITLE)

; Set item 1 to 10.
ControlSetText($TITLE, "", "Edit1", 10)
; Set item 2 to 20.
ControlSetText($TITLE, "", "Edit2", 20)
; Set item 3 to 30.
ControlSetText($TITLE, "", "Edit3", 30)
; Item 1 is taxable.
ControlClick($TITLE, "", "Button1")
; Item 3 is taxable.
ControlClick($TITLE, "", "Button3")

; Press Add button.
ControlClick($TITLE, "", "Button4")

; Get total.
$tot = ControlGetText(

; Verify total.
If $tot <> "62" Then
    ConsoleWriteError(
                & $tot &
EndIf
```

```
>"C:\Program Files\AutoIt3\SciTE
+>16:22:54 Starting AutoIt3Wrappe
>Running AU3Check (1.54.19.0)  f
+>16:22:54 AU3Check ended.rc:0
>Running:(3.3.6.1):C:\Program Fi
+>16:22:56 AutoIt3.exe ended.rc:
```

li=1 co=1 INS (CR+LF)

Item 1	10	[✓ Taxable?]
Item 2	20	Taxable?
Item 3	30	[✓ Taxable?]
Pretax	60	
Tax	2	
Total	62	

Figure 3: AutoIt.

4. AutoIt Overview

AutoIt is "a freeware BASIC-like scripting language designed for automating the Windows GUI and general scripting."(AutoIt, 2010) (When downloading, make sure to install both the "AutoIt Full Installation" and the "AutoIt Script Editor.")

The language is procedural but not object-oriented. Figure 4 shows an example of an AutoIt function that adds line numbers to a text string. Like PHP, AutoIt requires a dollar sign before each variable name. A comment starts with a semicolon and continues to the end of the line. A statement that continues to the next line must use an underscore at the end of the line as a continuation character. The functions `StringStripWS()`, `StringSplit()`, `UBound()`, `StringFormat()`, and `ConsoleWrite()` are from AutoIt's standard library. String concatenation uses the ampersand (&) symbol. The term @CRLF is a "macro" that signifies an end-of-line sequence of carriage return and line feed.

```
; Returns $text string with a line number
; at the beginning of each line.
Func NumberLines($text)

    ; Strip whitespace.
    $text = StringStripWS($text, 3)

    ; Break into lines (array of strings).
    $lines = StringSplit($text, @CRLF, 1)

    ; Build result string.
    $out = ""
    For $i = 1 To UBound($lines) - 1
        $out &= StringFormat("%d. %s\n", _
                $i, $lines[$i])
    Next
    Return $out

EndFunc

; Test the function.
$s = "first line" & @CRLF _
     & "second line" & @CRLF
ConsoleWrite(NumberLines($s))
```

Figure 4: An AutoIt function.

AutoIt has four different looping statements, including while, do-until, and two kinds of for statements. It also has if-else, select-case, and switch-case selection statements.

The extensive standard library includes many functions for starting and manipulating Windows programs. For example, the code in Figure 5 starts the Notepad text editor, waits for it to finish loading, and then enters some text into the text area.

AutoIt recognizes integer, floating-point, string, Boolean, binary, pointer, and variant types. The only built-in collection type is arrays, but because AutoIt supports COM (Component Object Model), it can also use collection types from .NET and Windows Script. Figure 6 shows an AutoIt program that uses an ArrayList from .NET.

AutoIt's support of COM also allows it to access external resources such as database systems. For example, Figure 7 shows how to query a Firebird relational database with SQL.

```
Run("notepad.exe")
WinWaitActive("Untitled - Notepad")
Send("This is some text.")
```

Figure 5: Start Notepad and enter text.

```
; Define a .NET ArrayList.
$class = "System.Collections.ArrayList"
$list = ObjCreate($class)

; Add elements.
$list.Add("Intel Corporation")
$list.Add("Hewlett-Packard")
$list.Add("General Motors")

; Iterate through the list.
For $company In $list
    ConsoleWrite($company & @CRLF)
Next
```

Figure 6: Using .NET within AutoIt.

5. System Testing Examples

Figure 8 shows a simple SUT: a Sales Total application that sums up to three items, each of which may be subject to a 5% sales tax. For student assignments, we typically include about three deliberate errors in the SUT for students to find.

We wrote the application in AutoIt using its Koda GUI utility and compiled it to an executable with AutoIt's Aut2Exe utility. We also used Aut2Exe's obfuscation option to thwart decompilation, so students cannot simply examine the source code to look for errors.

```
; Create a connection object.
$conn = ObjCreate("ADODB.Connection")

; Connect to the database.
$conn.Open( _
   "DRIVER=Firebird/InterBase(r) driver;" _
   & "DATABASE=C:\\sample.fdb;" _
   & "USER=SYSDBA;PWD=masterkey;")

; Query the database.
$rs = $conn.execute("SELECT Id, Name " _
   & "FROM Person ORDER BY Name")

; Display the results.
While Not $rs.EOF
   ConsoleWrite($rs.fields("Id").value _
       & ", " & $rs.fields("Name").value _
       & @CRLF)
   $rs.MoveNext()
WEnd
```

Figure 7: Accessing a Firebird database.

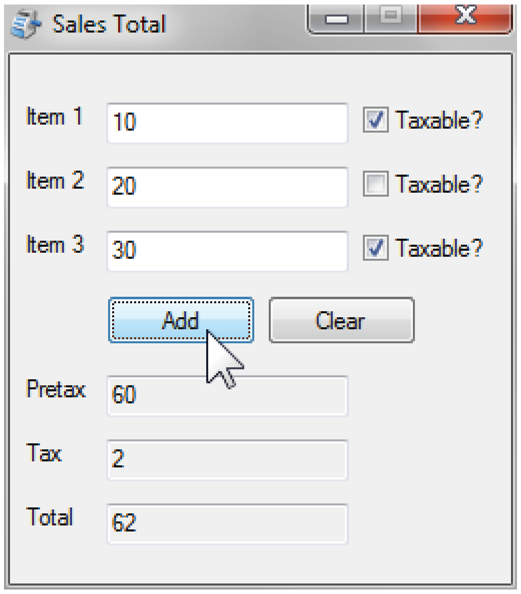


Figure 8: Sales Total desktop application.

Record/Playback Scripting

Students use AutoIt's record/playback utility for their first system-testing assignment. By using record/playback, students see that although record/playback seems to make system testing almost trivial, it has significant problems. (They realize this by the second assignment, described later.)

Students use AutoIt's AU3Recorder to record all mouse and keyboard activity while using the SUT. When the student finishes, AU3Recorder generates AutoIt code to reproduce the student's actions. The result is similar to code shown in Figure 9.

The code in Figure 9 starts the SUT, waits for it to load, enters test data into the fields, and clicks the Add button. Students can manually verify that the result is correct, but manual verification is tedious and error prone, particularly when running many test cases.

```
Run("SalesTotal")
WinWaitActive("Sales Total")
Send("10{TAB}{SPACE}{TAB}20{TAB}{TAB}30")
Send("{TAB}{SPACE}{TAB}{ENTER}")
```

Figure 9: Record/playback without verification.

Therefore, students must add code to Figure 9 to verify that the displayed results are correct. An easy way to capture the displayed result is to tab to the Total field, then copy the value into the clipboard by typing Control-C. The student runs AU3Recorder again, this time copying the total into the clipboard. The student then writes code to compare the contents of the clipboard with the expected value, as shown in Figure 10.

```
Run("SalesTotal")
WinWaitActive("Sales Total")
Send("10{TAB}{SPACE}{TAB}20{TAB}{TAB}30")
Send("{TAB}{SPACE}{TAB}{ENTER}")
Send("{TAB}{TAB}{TAB}{TAB}")
Send("{CTRLDOWN}c{CTRLUP}")

; Verify results (student-added code).
#include <ClipBoard.au3>
If _ClipBoard_GetData() <> "62" Then
    ConsoleWriteError("Expected 62, got ")
    ConsoleWriteError(_ClipBoard_GetData())
    ConsoleWriteError(@CRLF)
EndIf
```

Figure 10: Record/playback with verification.

Of course, one test case is not enough to test this SUT adequately, so students will need to repeat this process with combinations of taxable and nontaxable items, missing items, invalid entries, and so on. The result, which is typically many lines long, is highly sensitive to the layout of user interface controls. For example, if positions of the Add and Clear buttons are reversed, none of the tests will work correctly. Another problem with record/playback is that playback sends individual keystrokes and mouse movements to the SUT, which can be time consuming. After the student has added several more test cases, running these tests takes an inordinate amount of time.

For the second testing assignment, students use the same SUT but with minor changes to the user interface that break their record/playback scripts. Students thus experience the major disadvantage of using record/playback: test cases are extremely sensitive to changes in the user interface. We also fix some errors in the first version of the SUT and add a couple new ones.

We then show students how to access user interface controls directly from AutoIt, rather than by simulated keyboard and mouse activity. Our informal experiments show a speedup factor of about fifteen with direct access. (AutoIt's direct access only works with standard Windows controls, such as those found in Visual Studio. It does not work with nonstandard controls, such as those used in Delphi, QT, Java Swing, or Motif.)

As an example of an AutoIt direct access function, ControlSetText() inserts a text value directly into a text edit control. This standard function takes three parameters: the name of the SUT window, the Windows ID of the control, and the value to insert. AutoIt's AU3Info utility makes it easy to find the Windows ID of a control: simply move the mouse over the control to get its ID. For example, AU3Info reports that the Windows ID of the "Item 1" control in the Sales Total application is "Edit1."

Figure 11 shows how to use ControlSetText() and ControlClick() to insert values directly into the Sales Total application, then retrieve the total with ControlGetText().

Building a System Testing Framework

The approach taken in Figure 11 is simple and straightforward, but does not scale well. The single test case sprinkles its test data over several statements; when the code includes several test cases, it is not apparent whether the test cases are sufficient.

```
Const $TITLE = "Sales Total"

Run("SalesTotal")
WinWaitActive($TITLE)

; Set item 1 to 10.
ControlSetText($TITLE, "", "Edit1", 10)
; Set item 2 to 20.
ControlSetText($TITLE, "", "Edit2", 20)
; Set item 3 to 30.
ControlSetText($TITLE, "", "Edit3", 30)
; Item 1 is taxable.
ControlClick($TITLE, "", "Button1")
; Item 3 is taxable.
ControlClick($TITLE, "", "Button3")

; Press Add button.
ControlClick($TITLE, "", "Button4")

; Get total.
$tot = ControlGetText($TITLE, "", "Edit6")

; Verify total.
If $tot <> "62" Then
    ConsoleWriteError("Expected 62, got " _
        & $tot & @CRLF)
EndIf
```

Figure 11: Direct access.

Therefore, we build a system-testing framework so that writing test cases becomes trivial and the test data is obvious. We have found that students enjoy developing a system-testing framework collaboratively during in-class discussion.

An organizational scheme that we have found useful divides the framework into three files:

- A file of general system testing functions, such as StartSUT(), AssertEquals(), and AssertError() that apply to testing any desktop SUT (see Listing 1 in the appendix),
- A file of support functions specific to a particular desktop SUT, such as entering values into the SUT and verifying results (see Listing 2 in the appendix), and
- A file of test cases as function calls (see Listing 3 in the appendix).

We produced the first eleven test cases in Listing 3 using the *pairwise testing* approach (Cohen, Dalal, Parelius, & Patton, 1996) with the following values:

- For each item: blank, a whole number, and a number with a decimal point
- For each checkbox: True and False (always False when the corresponding item is blank)

The last three test cases insert an invalid value into each item, which should cause the SUT to generate errors.

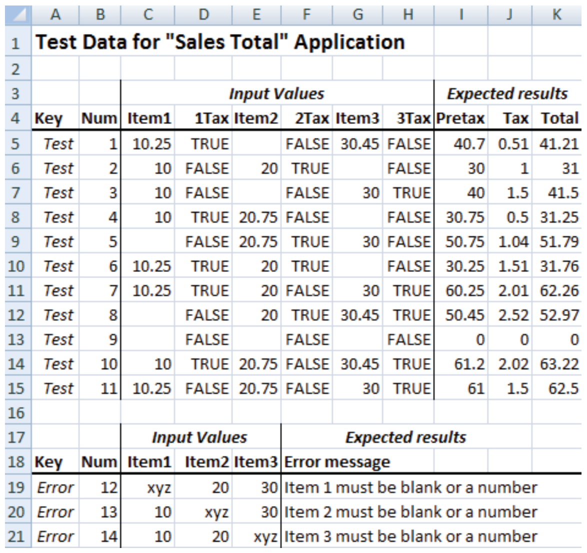
Each test case is simply a function call, which makes it easy to write and maintain test cases, because testers can concentrate on test cases and test data alone. The format of Listing 3 greatly simplifies verification that the tests cases include all pairs.

Storing Test Data in a Spreadsheet

With a little more work, a spreadsheet can store the test data in a clear and easy-to-use format, as shown in Figure 12. Besides clarity, another benefit of storing test data in a spreadsheet is that students can use formulas to compute expected results.

Figure 12 uses the *keyword-based* format (Nagle, 2010; Fewster & Graham, 1999). This format uses a keyword, typically in the first column, to indicate the kind of test to perform. For example, the keyword "Test" in row 4 indicates a normal test, while the keyword "Error" in row 17 indicates that the given test data should produce an error.

Listing 4 in the appendix shows the OpenSpreadsheet() function that opens an existing spreadsheet for reading. Listing 5 shows the SUT-specific code to read each row of the spreadsheet and call the appropriate SUT-specific function from Listing 2.



	A	B	C	D	E	F	G	H	I	J	K
1	**Test Data for "Sales Total" Application**										
2											
3			*Input Values*						*Expected results*		
4	**Key**	**Num**	**Item1**	**1Tax**	**Item2**	**2Tax**	**Item3**	**3Tax**	**Pretax**	**Tax**	**Total**
5	*Test*	1	10.25	TRUE		FALSE	30.45	FALSE	40.7	0.51	41.21
6	*Test*	2	10	FALSE	20	TRUE		FALSE	30	1	31
7	*Test*	3	10	FALSE		FALSE	30	TRUE	40	1.5	41.5
8	*Test*	4	10	TRUE	20.75	FALSE		FALSE	30.75	0.5	31.25
9	*Test*	5		FALSE	20.75	TRUE	30	FALSE	50.75	1.04	51.79
10	*Test*	6	10.25	TRUE	20	TRUE		FALSE	30.25	1.51	31.76
11	*Test*	7	10.25	TRUE	20	FALSE	30	TRUE	60.25	2.01	62.26
12	*Test*	8		FALSE	20	TRUE	30.45	TRUE	50.45	2.52	52.97
13	*Test*	9		FALSE		FALSE		FALSE	0	0	0
14	*Test*	10	10	TRUE	20.75	FALSE	30.45	TRUE	61.2	2.02	63.22
15	*Test*	11	10.25	FALSE	20.75	FALSE	30	TRUE	61	1.5	62.5
16											
17			*Input Values*			*Expected results*					
18	**Key**	**Num**	**Item1**	**Item2**	**Item3**	**Error message**					
19	*Error*	12	xyz	20	30	Item 1 must be blank or a number					
20	*Error*	13	10	xyz	30	Item 2 must be blank or a number					
21	*Error*	14	10	20	xyz	Item 3 must be blank or a number					

Figure 12: Test data in a spreadsheet.

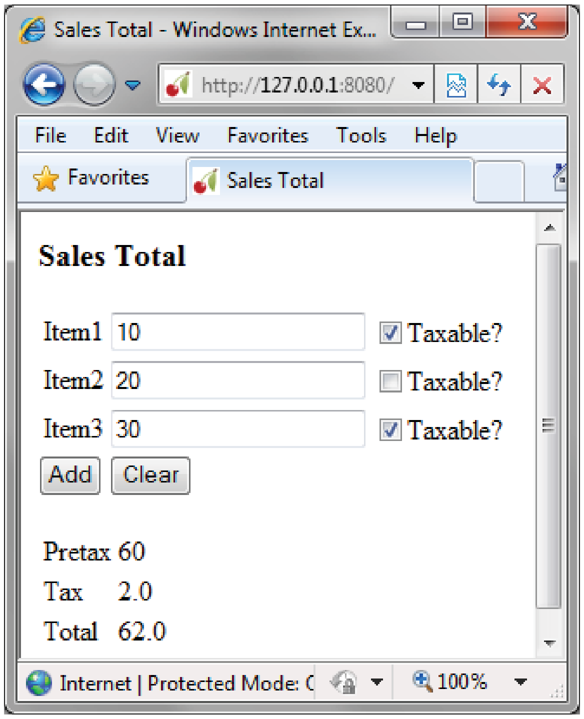
Web Application Testing

Testing web applications is conceptually the same as testing desktop applications, but can require more setup on the instructor's part. More setup is necessary because students need access to a web application they can install on their own computers (so students do not bog down a shared SUT with tests). Furthermore, the SUT source code must be inaccessible (so students look for errors by testing, not by examining the SUT source). Instructors need to develop web SUTs that either compile to executables or sufficiently obfuscate source code. The resulting web SUT must also be easy to distribute to students and easy for students to install on their computers.

Many approaches meet these requirements for developing web SUTs, and the best choice for a particular instructor depends on the instructor's familiarity with the programming language and tools used by that approach. For example, we teach Python and Java in our introductory programming courses, so we develop our web SUTs in those languages.

Figure 13 shows the Sales Total application converted to a web application using the CherryPy web framework (cherrypy.org, 2010), which uses Python. CherryPy is relatively easy to install and includes its own web server. We distribute CherryPy web applications as compiled Python bytecode to deter students from referring to the SUT source code.

AutoIt's standard library has an extensive collection of functions for accessing and manipulating the Internet Explorer web browser. It includes functions to read and write text on a web page, enter and read form controls, submit forms, follow links, and more. For example, the _IEFormElementCheckBoxSelect() function puts a checkmark in a checkbox.



Sales Total

Item1	10	☑ Taxable?
Item2	20	☐ Taxable?
Item3	30	☑ Taxable?

Add Clear

Pretax 60
Tax 2.0
Total 62.0

Figure 13: Sales Total web application.

Listing 6 in the appendix shows an example of a single test case for the Sales Total web SUT. Listing 6 is conceptually the same as the code in Figure 11 for the desktop version of Sales Total, and like Figure 11, it does not scale well. We follow the same approach for building a web system-testing framework as we did for desktop applications. That is, we create a file of functions for testing any web application, and another file of support functions for testing a specific SUT. We then put the actual test cases in either a third source file or a spreadsheet.

6. Conclusion

We have been pleased with our selection of AutoIt for system testing. Its VB-like programming language, its ability to test desktop and web applications, its excellent documentation and support, its IDE, and its large standard library make it an excellent, free stand-in for a professional testing tool. Using AutoIt gives students an experience similar to that of professional QA practitioners.

Students experience both the appeal and significant disadvantages of record/playback. They learn how to write higher-level testing functions, organize those functions into a system-testing framework using a keyword-based format that stores test data separately. Finally, they see how using a custom testing framework simplifies the design and implementation of test cases for both desktop and web applications.

Although AutoIt may not be suitable for industrial use (because it cannot access nonstandard desktop GUI controls), it provides an experience similar to using professional tools, and thus makes an ideal educational system-testing tool.

7. References

AutoHotKey. (2010). AutoHotKey. Retrieved from http://www.autohotkey.com/

AutoIt. (2010). AutoIt. Retrieved from http://www.autoitscript.com/autoit3/index.shtml

Badboy Software. (2010). Badboy. Retrieved from http://www.badboy.com.au/

Beizer, B. (1995). Black-Box Testing: Techniques for Functional Testing of Software and Systems. Wiley.

Buchmann, R. A., Arba, R., & Mocean, L. (2009). Black Box Software Testing Console Implemented with AutoIT. Proceedings of the International Conference on Knowledge Engineering, Principles and Techniques, KEPT2009. Cluj-Napoca (Romania).

cherrypy.org. (2010). CherryPy. Retrieved from http://www.cherrypy.org

Cohen, D. M., Dalal, S. R., Parelius, J., & Patton, G. C. (1996). The Combinatorial Design

Approach to Automatic Test Generation. IEEE Software , 13 (5), 83-88.

Fewster, M., & Graham, D. (1999). Software test automation: effective use of test execution tools. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co.

Gargoyle Software Inc. (2010). Retrieved from HtmlUnit: http://htmlunit.sourceforge.net/

Garousi, V., & Mathur, A. (2010). Current State of the Software Testing Education in North American Academia and Some Recommendations for the New Educators. 23rd IEEE Conference on Software Engineering Education and Training (pp. 89-96). IEEE.

Gold, R. (2010). Retrieved from HttpUnit: http://httpunit.sourceforge.net/

Hewlett-Packard Development Company. (2010). HP QuickTest Professional software. Retrieved from HP.com: https://h10078.www1.hp.com/cda/hpms/display/main/hpms_content.jsp?zn=bto&cp=1-11-127-24^1352_4000_100__

IBM. (2010). Rational Functional Tester. Retrieved from http://www-01.ibm.com/software/awdtools/tester/functional/

JUnit.org. (2010). Retrieved from JUnit.org: http://www.junit.org

Minnesota Revenue. (2009, a). 2009 K–12 Education Credit. Retrieved from http://www.taxes.state.mn.us/forms/m1ed.pdf

Minnesota Revenue. (2009, b). 2009 Minnesota Individual Income Tax Forms and Instructions. Retrieved from http://www.taxes.state.mn.us/taxes/individ/instructions/m1_inst.pdf

Nagle, C. (2010). Test Automation Frameworks. Retrieved from SAS Institute: http://safsdev.sourceforge.net/FRAMESDataDrivenTestAutomationFrameworks.htm

Pezzè, M., & Young, M. (2008). Software Testing and Analysis: Process, Principles, and Techniques. Hoboken, NJ: John Wiley & Sons, Inc.

SciTE. (2010). Retrieved from SciTE: http://www.scintilla.org/SciTE.html

Selenium Project. (2010). Selenium web application testing system. Retrieved from http://seleniumhq.org/

Telcordia. (2010). Applied Research at Telcordia. Retrieved from AR Greenhouse: http://aetgweb.argreenhouse.com/

Wall, T. (2008). Retrieved from Abbot Java GUI Test Framework: http://abbot.sourceforge.net/doc/overview.shtml

Wikipedia. (2010). AutoHotKey. Retrieved from http://en.wikipedia.org/wiki/Autohotkey

APPENDIX

Table 1: System-testing tool evaluation summary.

Feature	JUnit	Badboy	Selenium	AutoHotKey	AutoIt
Programmability	1	1	5	2	4
Record/playback	0	5	5	4	4
External resource access	5	2	5	3	4
Desktop GUI and web testing	3	0	0	4	4
Control information	0	4	4	4	4
Includes IDE	0	5	5	5	5
Creates GUI executables	2	0	0	4	4
Support	5	5	5	3	4
TOTAL	**16**	**22**	**29**	**29**	**33**

Listing 1: Testing.au3 (General system testing functions)

```
AutoItSetOption("MustDeclareVars", 1)

; Start the system under test (SUT) if not already running.
; $windowTitle: The window title of the SUT.
; $exeName: The name of the executable file.
; $windowText: Additional text that must appear in the SUT window (optional).
```

```
Func StartSUT($windowTitle, $exeName, $windowText = "")
    If Not WinExists($windowTitle, $windowText) Then
        Run($exeName)
    EndIf
    WinWait($windowTitle, $windowText)
    If Not WinActive($windowTitle, $windowText) Then
        WinActivate($windowTitle, $windowText)
    EndIf
    WinWaitActive($windowTitle, $windowText)
EndFunc   ;==>StartSUT

; Ensure that the given condition is true, otherwise log an error.
; $testName: The name of the current test case.
; $condition: The Boolean condition that should be true.
; $message: Optional, additional information to appear with the error.
Func Assert($testName, $condition, $message = "")
    If Not $condition Then
        LogError($testName, $message)
    EndIf
EndFunc   ;==>Assert

; Ensure that the expected value equals the actual, otherwise log an error.
; $testName: The name of the current test case.
; $expected: The expected value.
; $actual: The actcual value.
; $message: Optional, additional information to appear with the error.
Func AssertEquals($testName, $expected, $actual, $message = "")
    If $message <> "" Then
        $message = ": " & $message
    EndIf
    If $expected <> $actual Then
        Assert($testName, $expected == $actual, "Expected " & $expected _
                 & ", but found " & $actual & $message)
    EndIf
EndFunc   ;==>AssertEquals

; Ensure that a new error message box appears.
; $testName: The name of the current test case.
; $errorWindowTitle: The window title of the error message box.
; $ackButtonName: The name of the button control used to acknowledge the error.
; $errorMessage: The expected error message to appear in the message box (optional).
Func AssertError($testName, $errorWindowTitle, $ackButtonName, $errorMessage = "")
    WinWait($errorWindowTitle, "", 1)
    If Not WinActive($errorWindowTitle, "") Then WinActivate($errorWindowTitle, "")
    WinWaitActive($errorWindowTitle, "", 1)
    If WinExists($errorWindowTitle) Then
        If Not WinExists($errorWindowTitle, $errorMessage) Then
            LogError($testName, "Wrong error message, expected '" & $errorMessage _
                     & "', but found '" & ToOneLine(WinGetText($errorWindowTitle)) & "'")
        EndIf
        ControlClick($errorWindowTitle, "", $ackButtonName)
    Else
        LogError($testName, "Did not get any error, expected '" & $errorMessage & "'")
    EndIf
EndFunc   ;==>AssertError

; ----------------------------------------------------------------------------
; Internal functions.
; ----------------------------------------------------------------------------
; Report an an error (internal function).
; $testName: The name of the test case that failed.
; $message: The error message to log.
Func LogError($testName, $message)
    If $message <> "" Then
        $message = ": " & $message
    EndIf
    ConsoleWriteError("ERROR in test " & $testName & $message & @CRLF)
EndFunc   ;==>LogError

; Convert a multiline string to a single line.
; $string: The multiline string.
; Returns: The same string but all on one line.
Func ToOneLine($string)
    Return StringStripWS(StringReplace(StringReplace($string, Chr(10), " ") _
            , Chr(13), " "), 7)
EndFunc   ;==>ToOneLine
```

Listing 2: SalesTotalTesting.au3 (Support Functions for testing the "Sales Total" application)

```
#include "Testing.au3"

AutoItSetOption("MustDeclareVars", 1)

Dim Const $WINDOW_TITLE = "Sales Total"
Dim Const $ERROR_WINDOW_TITLE = "Sales Total Error"

; Clicks the Clear button.
Func ClickClearButton()
    ControlClick($WINDOW_TITLE, "", "Button5")
EndFunc   ;==>ClickClearButton

; Clicks the Add button.
Func ClickAddButton()
    ControlClick($WINDOW_TITLE, "", "Button4")
EndFunc   ;==>ClickAddButton

; Enters values into the application, without pressing a button.
; $item1: The cost of the first item.
; $item1Taxable: If true, item 1 is taxable.
; $item2: The cost of the second item.
; $item2Taxable: If true, item 2 is taxable.
; $item3: The cost of the third item.
; $item3Taxable: If true, item 3 is taxable.
Func EnterValues($item1, $item1Taxable, $item2, $item2Taxable, $item3, $item3Taxable)
    ControlSetText($WINDOW_TITLE, "", "Edit1", $item1)
    ControlSetText($WINDOW_TITLE, "", "Edit2", $item2)
    ControlSetText($WINDOW_TITLE, "", "Edit3", $item3)
    If $item1Taxable Then
        ControlClick($WINDOW_TITLE, "", "Button1")
    EndIf
    If $item2Taxable Then
        ControlClick($WINDOW_TITLE, "", "Button2")
    EndIf
    If $item3Taxable Then
        ControlClick($WINDOW_TITLE, "", "Button3")
    EndIf
EndFunc   ;==>EnterValues

; Ensures that the results are as expected.
; $testName: The name of the currently running test case.
; $pretax: The expected pretax value.
; $tax: The expected tax value.
; $total: The expected total value.
Func VerifyResults($testName, $pretax, $tax, $total)
    AssertEquals($testName, $pretax, ControlGetText($WINDOW_TITLE, "", "Edit4"), "Pretax")
    AssertEquals($testName, $tax, ControlGetText($WINDOW_TITLE, "", "Edit5"), "Tax")
    AssertEquals($testName, $total, ControlGetText($WINDOW_TITLE, "", "Edit6"), "Total")
EndFunc   ;==>VerifyResults

; Runs a testcase by entering the given values, pressing the Add button, and
; ensuring that the results equal the given expected values.
; $testName: The name of the currently running test case.
; $item1: The cost of the first item.
; $item1Taxable: If true, item 1 is taxable.
; $item2: The cost of the second item.
; $item2Taxable: If true, item 2 is taxable.
; $item3: The cost of the third item.
; $item3Taxable: If true, item 3 is taxable.
; $pretax: The expected pretax value.
; $tax: The expected tax value.
; $total: The expected total value.
Func RunTest($testName, $item1, $item1Taxable, $item2, $item2Taxable, $item3, _
        $item3Taxable, $pretax, $tax, $total)
    ClickClearButton()
    EnterValues($item1, $item1Taxable, $item2, $item2Taxable, $item3, $item3Taxable)
    ClickAddButton()
    VerifyResults($testName, $pretax, $tax, $total)
EndFunc   ;==>RunTest

; Runs a testcase by entering the given values, pressing the Add button, and
; ensuring that an error message box appears with the given message.
; $testName: The name of the currently running test case.
; $item1: The cost of the first item.
; $item1Taxable: If true, item 1 is taxable.
; $item2: The cost of the second item.
; $item2Taxable: If true, item 2 is taxable.
; $item3: The cost of the third item.
; $item3Taxable: If true, item 3 is taxable.
; $expectedMessage: The error message that should appear.
Func RunTestError($testName, $item1, $item2, $item3, $expectedMessage)
    ClickClearButton()
    EnterValues($item1, False, $item2, False, $item3, False)
    ClickAddButton()
    AssertError($testName, $ERROR_WINDOW_TITLE, "Button1", $expectedMessage)
EndFunc   ;==>RunTestError

; Starts the Sales Total application if it is not already running.
Func StartSalesTotal()
    StartSUT($WINDOW_TITLE, "SalesTotal")
EndFunc   ;==>StartSalesTotal
```

Listing 3: SalesTotalTestCases.au3 (Test cases for the "Sales Total" desktop application)

```
#include "SalesTotalTesting.au3"

AutoItSetOption("MustDeclareVars", 1)

StartSalesTotal()
; Tests:   Item1   1Tax   Item2  2Tax   Item3  3Tax   Pretax Tax   Total
RunTest(1, 10.25,  True,  "",    False, 30.45, False, 40.7,  0.51, 41.21)
RunTest(2, 10,     False, 20,    True,  "",    False, 30,    1.00, 31.00)
RunTest(3, 10,     False, "",    False, 30,    True,  40,    1.50, 41.50)
RunTest(4, 10,     True,  20.75, False, "",    False, 30.75, 0.50, 31.25)
RunTest(5, "",     False, 20.75, True,  30,    False, 50.75, 1.04, 51.79)
RunTest(6, 10.25,  True,  20,    True,  "",    False, 30.25, 1.51, 31.76)
RunTest(7, 10.25,  True,  20,    False, 30,    True,  60.25, 2.01, 62.26)
RunTest(8, "",     False, 20,    True,  30.45, True,  50.45, 2.52, 52.97)
RunTest(9, "",     False, "",    False, "",    False, 0,     0,     0.00)
RunTest(10, 10,    True,  20.75, False, 30.45, True,  61.2,  2.02, 63.22)
RunTest(11, 10.25, False, 20.75, False, 30,    True,  61,    1.50, 62.50)
; Errors:        Item1  Item2  Item3  Expected message
RunTestError(12, "xyz", 20,    30,    "Item 1 must be blank or a number")
RunTestError(13, 10,    "xyz", 30,    "Item 2 must be blank or a number")
RunTestError(14, 10,    20,    "xyz", "Item 3 must be blank or a number")
```

Listing 4: SpreadsheetTest.au3 (Support functions for storing test data in a spreadsheet)

```
#include<Excel.au3>

; Returns an Excel spreadsheet with the given title and path.  If the spreadsheet is
; already open in Excel, it returns that spreadsheet, otherwise, it opens the
; spreadsheet.
; $title: The title of the spreadsheet.
; $path: The absolute path (file location) of the spreadsheet.
; Returns: The spreadsheet with the given title and path.
Func OpenSpreadsheet($title, $path)
    Local $oExcel
    If WinExists($title, "") Then
        $oExcel = _ExcelBookAttach($path)
    Else
        $oExcel = _ExcelBookOpen($path)
    EndIf
    If @error <> 0 Then
        MsgBox(0, "Error!", "Unable to open the Excel spreadsheet " & $path)
        Exit
    EndIf
    Return $oExcel
EndFunc   ;==>OpenSpreadsheet
```

Listing 5: SalesTotalExcel.au3 (Run "Sales Total" test cases from an Excel spreadsheet)

```
AutoItSetOption("MustDeclareVars", 1)

#include "SpreadsheetTesting.au3"
#include "SalesTotalTesting.au3"

Global Const $EXCEL_PATH = @WorkingDir & "\SalesTotalTestData.xlsx"
Global Const $EXCEL_TITLE = "Microsoft Excel - SalesTotalTestData.xlsx"

Func RunTests($testData)
    For $row = 1 To $testData[0][0] ; row count
        Local $keyword = $testData[$row][1]
        Switch $keyword
            Case "Test"
                Local $testName = $testData[$row][2]
                Local $item1Value = $testData[$row][3]
                Local $item1Taxable = $testData[$row][4]
                Local $item2Value = $testData[$row][5]
                Local $item2Taxable = $testData[$row][6]
                Local $item3Value = $testData[$row][7]
                Local $item3Taxable = $testData[$row][8]
                Local $pretax = $testData[$row][9]
                Local $tax = $testData[$row][10]
                Local $total = $testData[$row][11]
                RunTest($testName, $item1Value, $item1Taxable, $item2Value, _
                        $item2Taxable, $item3Value, $item3Taxable, $pretax, $tax, $total)
            Case "Error"
                Local $testName = $testData[$row][2]
                Local $item1Value = $testData[$row][3]
                Local $item2Value = $testData[$row][4]
                Local $item3Value = $testData[$row][6]
                Local $expectedMessage = $testData[$row][7]
                RunTestError($testName, $item1Value, $item2Value, $item3Value, _
                        $expectedMessage)
        EndSwitch
    Next
EndFunc   ;==>RunTests

Global $oExcel = OpenSpreadsheet($EXCEL_TITLE, $EXCEL_PATH)
Global $testData = _ExcelReadSheetToArray($oExcel)
StartSalesTotal()
RunTests($testData)
```

Listing 6: SalesTotalWebTest.au3 (Simple example of single web test case)

```
#include <IE.au3>

; Open the site.
$browser = _IECreate("http://127.0.0.1:8080")

; Get the form.
$form = _IEFormGetObjByName($browser, "salesform")

; Set item 1 to 10.
$item1String = _IEFormElementGetObjByName($form, "item1String")
_IEFormElementSetValue($item1String, "10")
; Item 1 is taxable.
_IEFormElementCheckBoxSelect($form, "item1Taxable")
; Set item 2 to 20.
$item2String = _IEFormElementGetObjByName($form, "item2String")
_IEFormElementSetValue($item2String, "20")
; Set item 3 to 30.
$item3String = _IEFormElementGetObjByName($form, "item3String")
_IEFormElementSetValue($item3String, "30")
; Item 3 is taxable.
_IEFormElementCheckBoxSelect($form, "item3Taxable")
; Get the Add button (0 = first button).
$addButton = _IEFormElementGetObjByName($form, "button", 0)

; Submit the form.
_IEAction($addButton, "click")

; Verify results.
$pretaxSumObject = _IEGetObjById($browser, "pretaxSum")
$pretaxSum = _IEPropertyGet($pretaxSumObject, "innertext")
If $pretaxSum <> "60" Then
    ConsoleWriteError('Error: pretax sum expected 60, got: ' & $pretaxSum & @CRLF)
EndIf
$taxSumObject = _IEGetObjById($browser, "taxSum")
$taxSum = _IEPropertyGet($taxSumObject, "innertext")
If $taxSum <> "2.0" Then
    ConsoleWriteError('Error: tax sum expected 2.0, got: ' & $taxSum & @CRLF)
EndIf
```

```
$totalSumObject = _IEGetObjById($browser, "totalSum")
$totalSum = _IEPropertyGet($totalSumObject, "innertext")
If $totalSum <> "62.0" Then
    ConsoleWriteError('Error: total sum expected 62.0, got: ' & $totalSum & @CRLF)
EndIf
```