System Testing on the Cheap

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 testcases 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 testcases directly from specifications. An example specification might be that the application is password protected. A system-level testcase 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

Garousi & Mathur (2010) 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."*

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, & Mocean (2009) used AutoIT to develop an elegant GUI testcase execution program that reads testcase information from a text file. For each testcase, 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 widgets and even images.

©2010 EDSIG (Education Special Interest Group of the AITP)

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 virtually all 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 of these major application areas, ideally Windows GUIs and AJAX on the web.

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.

Widget information: The tool should include the ability to find input and output widgets 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, therefore it is not suitable 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 excels at record/playback. Badboy's integrated help file includes several well-written tutorials.

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.

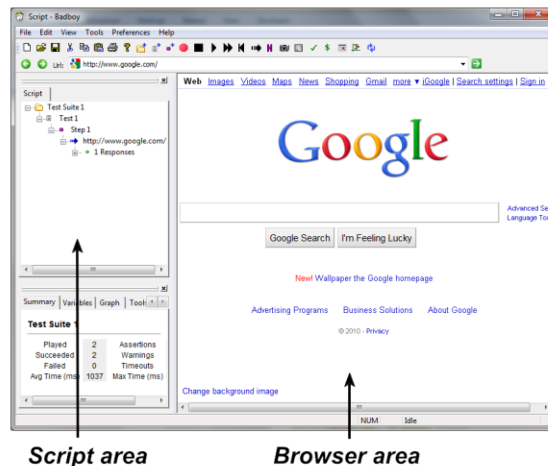

Figure 1: Badboy.

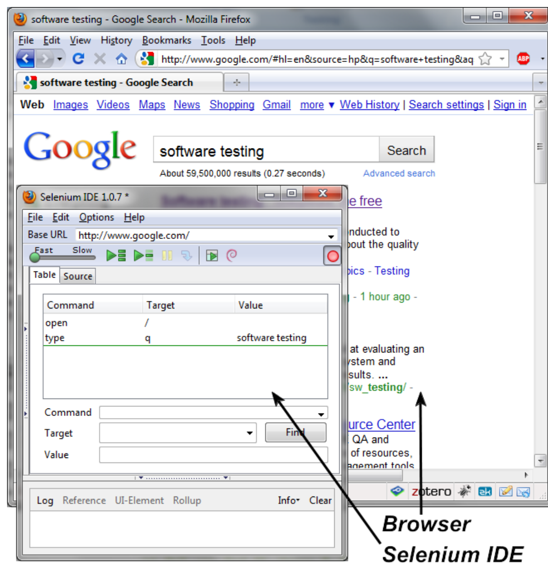

Figure 2: Selenium.

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 both 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.

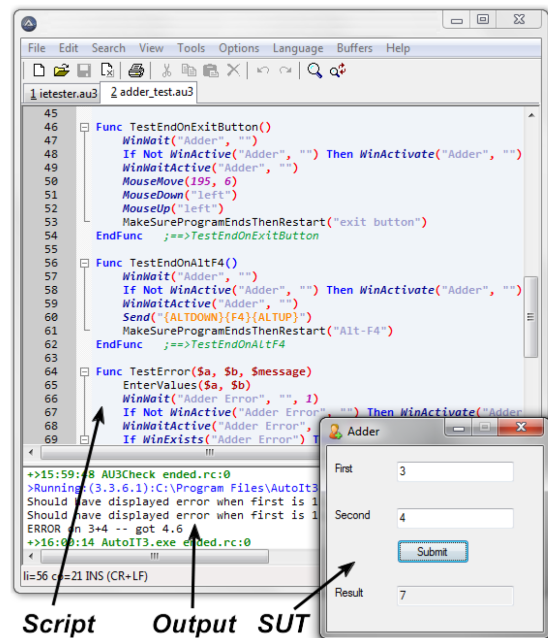

Figure 3: AutoIT.

Table 1 (in the appendix) summarizes the author's subjective evaluation of the testing tools we considered. The rating scale goes from 0

(not present) to 5 (excellent support). AutoIT emerges as the clear winner in this evaluation.

4. EXAMPLE ASSIGNMENTS

Like Beizer, we found that income tax documents provide an abundance of useful application ideas (Beizer, 1995). We developed TaxCalc, a simple desktop GUI application that computes deductions for educational expenses. Figures 5 and 4 show the application's specification, which (along with accompanying instructions) comprised the entirety of what students could refer to for writing testcases. In particular, they had no access to the program's source code.

We wrote TaxCalc using AutoIT's Koda GUI utility, and converted it to an executable with AutoIT's Aut2Exe utility. Aut2Exe has an obfuscation option that thwarts decompilation. Figure 6 shows the pseudocode of TaxCalc version 1, which was not available to students.

14 Personal computer hardware and educational software expenses, not to exceed $200. *(Do not include monthly service fees for Internet access)* .... _____

15 Add line 13 and line 14 .... _____

16 Multiply line 15 by 75% (.75) .... _____

17 If your household income on line 6 is:
- $33,500 or less, multiply the number of qualifying children in grades K-12 by $1,000
- More than $33,500, complete the worksheet on back .... _____

18 Amount from line 16 or line 17, whichever is less.
Full-year residents: Also enter this amount on line 29 of Form M1 .... _____

Figure 4: Schedule M1ED.
(Minnesota Revenue, 2009)

We produced several versions of TaxCalc with slightly different user interfaces and intentional errors.

Manual testing

Students tested the first version of TaxCalc, shown in Figure 7, manually, with the vague task to find as many errors as they could.

Manual testing can have tremendous value, such as when the tester is not familiar with the SUT. Because this was the first time students used TaxCalc, manual testing was particularly appropriate for this first assignment.

Worksheet for line 7, *K-12 education expense subtraction*

If you qualify for the K–12 education credit (line 29 of Form M1), and you cannot use all of your education expenses on Schedule M1ED, complete the following steps to determine line 7 of Form M1:

1 Qualifying tuition expenses . . ________
2 Qualifying computer expenses in excess of $200, up to a maximum of $200 . . . . ________

Complete steps 3–6 if on Schedule M1ED line 17 is less than line 16.

3 Line 15 of Schedule M1ED . . . ________
4 Line 18 of Schedule M1ED . . . ________
5 Multiply step 4 by 1.333 . . . . . ________
6 Subtract step 5 from step 3 . . . ________
7 Add steps 1, 2 and 6 . . . . ________

Figure 5: Educational expense calculation.
(Minnesota Revenue, 2009, p. 10)

```
input computer, tuition, line15, line16,
      line17, line18
if line17 < line16 then
    step5 = line18 * 1.333
    step6 = line15 - step5
else
    step6 = 0
end
return tuition + computer + step6
```

Figure 6: Initial TaxCalc pseudocode.

However, the underlying motivation for the first assignment was to reinforce the importance of test automation. Students did not know that subsequent assignments would also use TaxCalc, and that automated tests are much easier to repeat.

Several students found all intentional errors, and most pointed out that the application should compute values of lines 16 and 18.

Tax Calculator
File Help
Qualifying tuition expenses 204
Qualifying computer expenses 25
Line 15 of Schedule M1ED 337
Line 16 of Schedule M1ED 252.75
Line 17 of Schedule M1ED 2000
Line 18 of Schedule M1ED 252.75
Submit Clear
Education expense subtraction is $229.00

Figure 7: TaxCalc version 1.

Simple Record/Playback Scripting

The second version of TaxCalc, shown in Figure 8, incorporated student suggestions to remove inputs for lines 16 and 18, and introduced new errors. Figure 9 shows the (correct) pseudocode for this and all subsequent versions of TaxCalc.

Form1
File Help
Qualifying tuition expenses 204
Qualifying computer expenses 25
Line 15 of Schedule M1ED 337
Line 17 of Schedule M1ED 2000
Submit Clear
Education expense subtraction 229.00

Figure 8: TaxCalc version 2.

The second assignment was to use AutoIT's record/playback utility to record a complete set of tests, so that students can test subsequent versions by simply "pressing a button." Students needed to add assertions to the resulting script to make testcases, and they were encouraged to write functions to make the testcases easier to maintain. For example, Figure 10 shows a script that AutoIT recorded. The absolute mouse coordinates and assumed tab ordering make the script extremely sensitive to user interface changes.

```
input computer, tuition, line15, line17
line16 = line15 * 0.75
line18 = min(line16, line17)
if line17 < line16 then
    step5 = line18 * 1.333
    step6 = line15 - step5
else
    step6 = 0
end
return tuition + computer + step6
```

Figure 9: Revised TaxCalc pseudocode.

```
MouseClick("left",212,74,1)
Send("204{TAB}25{TAB}337{TAB}2000")
MouseClick("left",51,247,1)
MouseMove(258,285)
MouseDown("left")
MouseMove(192,294)
MouseUp("left")
Send("{CTRLDOWN}c{CTRLUP}")
; Student must add assertion manually.
If ClipGet() <> 229 Then
    ConsoleWrite("ERROR")
EndIf
```

Figure 10: Example record script.

In contrast, Figure 11 shows an example of a high-level function that isolates user interface details in other functions, making the tests more robust.

Version 3 of TaxCalc, shown in Figure 12, changed the user interface just enough so that students who heeded the advice to use functions did not need to modify their scripts nearly as much.

Building a Testing Framework

In the next assignment, students converted their testcases to *keyword-based* format (Nagle, 2010; Fewster & Graham, 1999). This format stores test data in a spreadsheet, file, or database – separate from the high-level test functions. Students used an Excel spreadsheet to store the test data, because AutoIT supports Excel directly (see Figure 13).

The keyword-based format required students to write code to read each spreadsheet row, then call the appropriate AutoIT function from

©2010 EDSIG (Education Special Interest Group of the AITP)

the given keyword and parameters. Students also wrote code to insert the test results back into the spreadsheet, to the right of the parameters, pushing earlier test results to the right. This assignment showed students how to write a simple testing framework (with rudimentary reporting capability) from scratch.

```
Func TestCompute($expected, $tuition, _
        $computer, $line15, $line17)
    EnterValues($tuition, $computer, _
        $line15, $line17)
    $actual = GetResult()
    If $actual <> $expected Then
        Error($actual, $expected)
    EndIf
EndFunc
...
TestCompute(229, 204, 25, 337, 2000)
```

Figure 11: Example high-level function.

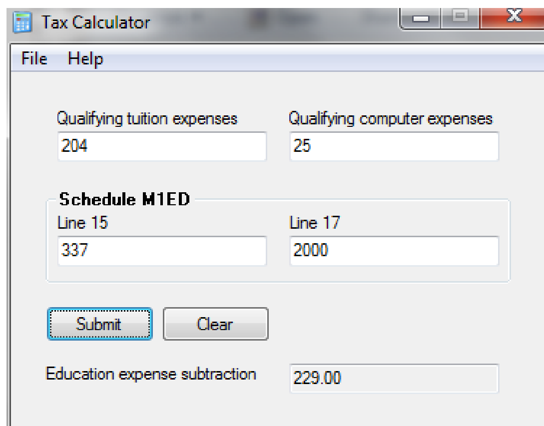

Figure 12: TaxCalc version 3.

Using an Existing Framework

The next AutoIT assignment required students to test a web-based version of TaxCalc, as shown in Figure 14.

In this assignment, students used a simple AutoIT framework created specifically for web testing. This framework includes AutoIT functions and spreadsheet keywords for:

- Navigation: following links, going to absolute web addresses,
- Form handling: setting and getting text parameters, setting checkboxes, submitting forms,
- Assertions for content by name and id,
- Simple database actions and assertions,
- Macros for defining new keywords from existing keywords

	A	B	C	D	E	F
1	**Keyword**	**Expected**	**Tuition**	**Computer**	**Line15**	**Line17**
2						
3	*Compute*	229	204	25	337	2000
4	*Compute*	799	640	159	209	1000
5	*Compute*	0	0	0	0	0
6	*Compute*	55	12	43	88	438
7						
8	*Missing*	Tuition		25	337	2000
9	*Missing*	Computer	204		337	2000
10	*Missing*	Line 15	204	25		2000
11	*Missing*	Line 17	204	25	337	
12						
13	*Invalid*	Tuition	hello	25	337	2000
14	*Invalid*	Computer	204	hello	337	2000
15	*Invalid*	Line 15	204	25	hello	2000
16	*Invalid*	Line 17	204	25	337	hello

Figure 13: Keyword-based format.

Because the framework was already finished, students just had to enter test data into the spreadsheet. This assignment showed students that a custom framework simplifies the design and implementation of actual testcases.

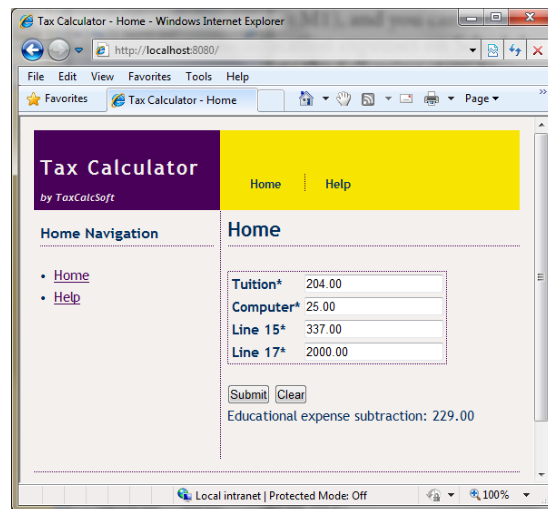

Figure 14: TaxCalc version 4.

In the next assignment, students used the same AutoIT testing framework to test a new job board application, shown in Figure 15. Unlike the assignments so far, the job board application used a database. Students used database actions and assertions in the framework to ensure the application changed the database appropriately. Figure 16 shows a small sample of testcases using this framework.

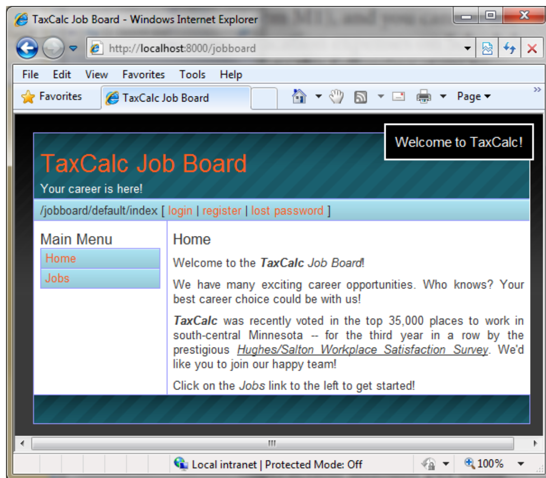

Figure 15: TaxCalc Job Board.

5. 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 a QA practitioner.

	A	B	C	D
1	**TaxCalc Job Board Test Cases**			
2				
3	**Test Case Name**	**Step Name**	**Step Arguments**	
44	**Login bad password**	*login*	admin@admin.admin	badpassword
45		*assertValueById*	flash-message	Invalid login
46				
47	**Login empty**	login		
48		*assertValueById*	email__error	Invalid email
49				
50	**Protected pages**	*assertNoLink*	Jobs & Applicants	
51		*assertNoAccess*	default/admin	
52		*assertNoAccess*	default/admin_add_job	
53		*assertNoAccess*	default/admin_job	
54		*assertNoAccess*	default/admin_applicant	
55				
56	**Login invalid email**	*login*	asdf	
57		*assertValueById*	email__error	Invalid email
58				
59	**Login invalid name**	*login*	wrong@wrong.wrong	
60		*assertValueById*	flash-message	Invalid login
61				
62	**Correct login**	*login*	admin@admin.admin	admin
63		*#assertValueById*	user-name	Admin

Figure 16: Job Board testcases.

Students have experienced both the appeal and significant disadvantages of record/playback. They learned how to write higher-level testing functions, and how to convert those functions into a keyword-based format that stores test data separately. Finally, they have seen how using a custom testing framework simplifies the design and implementation of testcases.

Although AutoIT may not be suitable for industrial use, it provides a similar feel, and thus makes an ideal educational system-testing tool.

6. 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).

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

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

1. APPENDIX

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
Widget 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**

Table 1: System-testing tool evaluation summary.