Standards for Test Automation

Brian Tervo
Windows XP Automation Applications Compatibility Test Lead
Microsoft Corporation

Overview

Over the last five years, I've had the opportunity to work in a group who writes automated tests using retail Windows applications to ensure compatibility in future releases of Windows, and to find bugs in the operating system. During that time, we have transitioned from having no standards to having a strict set of guidelines to follow when writing new automated tests. We have seen many benefits as a result of developing standards. In this paper, I will talk about these three areas:

1. Why there is a need for standards in test automation
2. Things to consider when developing test automation standards
3. Methods to ensure everyone adopts and follows the standards once they're written

Why there is a need for standards in test automation

I will start with a hypothetical situation. Suppose a company has decided to hire a team of testers to write test automation. They start by hiring two testers, Sandy and Joe. Both of these testers are given a test tool in a shrink-wrapped box, and asked to start writing automated test cases. The testers work independently, learn about the test tool and develop their own library of routines. They start implementing some test cases with automation and begin to see some success.

About a month later, the company hires two more testers, James and Sue. The manager of the team asks Sandy to mentor James and Joe to mentor Sue. Sandy shows James her library of routines. James completely adopts Sandy's framework, and eventually starts adding some of his own routines to Sandy's library files. Likewise, Joe shows Sue his library of routines. Although Sue finds most of the routines in Joe's library useful, there are some fundamental aspects of his framework that she does not like. After a couple of weeks, she makes copies of Joe's library files and starts modifying them to her suit her coding style.

A few more weeks pass, and the need for more automated tests increases, so the company allocates funds to have two more testers to write automation. They transferred Stephanie from within the company to the automation team because she has recently attended some courses on how to write test automation, and wanted the opportunity to use what she had learned. After glancing through the library files others on the team have created, she realized that others on the team made some fundamental mistakes based on what she learned in her classes. As a result, she wrote yet another new set of library functions.

Shortly after Stephanie joined the team, the company hired Bob who just graduated with a Computer Science Degree. Bob was a little disappointed with this job since he really wanted to be a developer and was upset that he had to resort to working in a test group. When Bob reviewed the library files from others on the team, he decided that none of the libraries was well organized or well structured. Bob created his own library of routines, which used linked lists, structures, pointers, and several other programming elements that the test tool was not designed to handle.

More months passed, and thousands of lines of test code were written. Since each tester was responsible for running the code they wrote on a regular basis, they found themselves spending their entire time running and triaging their tests, with no time left over to write new tests. To get the testers back to writing more automation, the company decided to move all of the tests to a lab and hire a lab manager to run the tests using a test harness.

Moving tests to a lab turned out to be a very expensive process. All of the libraries and several of the test scripts had to be modified so they could be run with the test harness. Several of the scripts had dependencies, such as screen resolution, environment variables, disk space, presence of certain drive letters, etc. that could not always be replicated in the lab environment. The dependencies varied from one library to the next. To move all of the tests to a single lab, many compromises had to be made, and each tester had to spend several weeks adapting their individual libraries to work in the lab.

The group finally got most of their automated tests running in a lab environment, shipped their product a couple months later, and started testing the next version of the product. Since most of the test cases for the old version of the product could still be run on the new version, the company chose to keep their automation team, and to continue writing more test cases for features being added to the new version. There were, however, a couple of major changes that happened shortly after the product shipped.

Bob accepted a job as a developer in another part of the company. The ownership of his code was distributed equally among the remaining members on the team. Unfortunately, nobody on the team was able to understand or Bob's code because of its complexity. They each spent many hours trying to determine the causes for script failures, and eventually came to the conclusion that their time would be better spent rewriting Bob's test cases using their own libraries. In effect, several months of Bob's work was lost because he was the only one who could fully understand how his code worked.

They encountered another problem when the new version of the product added a new feature that broke a major function in the test tool. After pleading with program managers and developers to remove the feature from the product, they realized that was a losing battle. They were going to have to find a way to work around the issue in their test tool. Stephanie was the first to come up with a solution in her library, and she showed her new function to the group. The rest of the group was excited about her innovation, and tried to call her library function directly. However, there were problems with doing that. Stephanie's function had dependencies on several of the other files in her library. If others on the team wanted to use her function, they had to include all of Stephanie's library files. This led to additional problems when Stephanie's libraries happened to use the same global variable names and function names as files in different libraries. As a result, nobody could use Stephanie's code directly, and they ended up having to copy and paste her code into their own library files and maintain it separately.

I will end this story here. It illustrates some of the common problems associated with not having standards for test automation.

1. *When global changes are required, everyone has to make the changes.* In this story, this happened twice. The first time was when the team decided they needed to run their tests under a harness, and code had to be added to the libraries to support the harness. If they had a single library, they would only need to modify one library rather than several libraries. The second example was when Stephanie wanted to share code from her library, but couldn't because of dependencies with her other library files. If they had a single library, Stephanie's new function would have been immediately available to everyone in the group.
2. *Tests are limited to run in the environment in which they were written*. This first became an issue for this test team when they wanted to move their tests to a lab. They learned that their tests had dependencies, and that these dependencies varied from one library to the next. If they had a standard machine configuration, or took the time to find out which dependencies needed to be handled by the test automation, the transfer of tests to a lab environment could have gone much smoother.
3. *Tests can only be debugged or maintained by the author*. When Bob left the group, he left behind several months of work that could not be used, because nobody else in the group could understand how his code worked. If the team had standards on coding structure and style, everyone would be able to read each other's tests, and Bob's code would not have been lost when he left the team.

Things to consider when developing test automation standards

It's important to look at the entire scope of your automation project when you create standards for test automation. If you overlook something while drafting the standards, it can be expensive to change standards after your team has created several thousand lines of code. Here is a list of some of the things you should consider when creating standards for test automation:

1. *Test Tools.* There are many automation test tools on the market. Some of them were designed for testing a specific class of applications. For example, some tools were written with the intent of testing GUI applications, and have functionality to manipulate objects commonly found in the UI of the operating system. Other tools were written to test Web-based applications and have built-in functions to manipulate objects commonly found on Web pages. Most test tools have both a scripting language and capture/playback tools. However, some are intended as scripting tools and to create tests by writing code, while others were designed to be primarily for capture/playback. Chances are you will use both the capture/playback and the scripting language features, regardless of which tool you choose. The capture/playback features of scripting tools can often be used to obtain code samples to learn more about the scripting language. But some amount of time is almost always required to tweak the recorded scripts to get them to run reliably and return viable pass/fail information during playback. All test tools have benefits and drawbacks, and it's beyond the scope of this paper to discuss all the details of what

each test tool does or does not offer. For further information on how to select a test automation tool, I recommend reading “Making the Right Choice” and “Evaluating Tools” by Elisabeth Hendrickson, listed in the references section at the end of this paper.

2. *Libraries*. I recommend that each project have just one library and that there is no duplication of code or functionality in the library. For example, if there is a library file with that already contains several functions to create random numbers, all random number functions in the library should be in that file. If a tester needs some new feature for generating random numbers, either an existing function in the library should be modified in a way that will not break any script currently using it, or a new function should be added to the random number function library file. Having just one library of routines rather than several will save time in library maintenance.
3. *Logging*. Having a common logging format is fundamental for automation that can be understood and debugged by everyone on the team. Consider what the log files should look like. Should they be in plain text or HTML? Should information be logged to a database? Where will the log files be stored? What information should be logged? At a quick glance, will any tester on the team be able to review a log from a test, determine whether the test passed or failed, and if necessary triage a failure? Will people who are not familiar with the automation be able to read the log files and understand the results? Does the test tool have built in logging functionality? If so, will it meet your needs, or will it be necessary to write a custom logging engine?
4. *Error Handling*. What should the tests do when they encounter an error? Should they log pertinent information about the failure, attempt to clean up, and move on to the next test? Or, should they stop everything, leaving the software in the failure state until someone investigates the failure? Should there be different levels of error handling, where the error is handled differently based on the severity of the failure?
5. *Environment*. What is the default test environment? For example, will all tests run at only one screen resolution and color depth? Or, will the tests have to run with several different resolutions and color depths? If so, which ones? How many logical drives are on the test machine? Will all the machines have the same number of drives with the same letter assignments? Will all drives with the same letter have the same file system? How much free space will be on each drive? Which operating system(s) will the tests have to run on? Which language versions of the software will be supported? Will the application under test be installed before any tests begin? If so, where will it be installed? If not, how will each script detect that the application is not installed and install it? Can each individual script assume there will always be a printer installed on the test machine? Which environment settings will the scripts be responsible for setting? Which settings can the scripts optionally change, and must they be returned to the default setting if the script ends? What happens if the script stops running before it changes things back to the defaults? Are there library functions available to set and restore each environment setting?
6. *Test Harnesses*. What information should the harness provide the test script about the application and the environment? What information should the test script provide the harness? What additional code, if any is required for the tests to interface with a test harness? Can any of this code be included in a library so it’s transparent to each individual script?

7. *Initialization and Clean-up Routines*. I recommend having a library function that is called at the beginning and another function that's called at the end of every script. There will likely be situations where there is a need to make changes prior to running any test, and having an initialization routine called by every script makes this possible. Clean-up routines in the library may be used to restore environment settings that an individual script failed to restore – possibly because the script encountered a failure it could not recover from.
8. *Unattended Running*. Should there be a requirement that scripts must run unattended (without any user interaction once the test has started)? If this requirement is in place, and the test script needs information from the user, can the test read the information from a configuration file that may be edited by the user prior to running the test? Or, can the test obtain this information from the test harness? If user interaction is allowed, under what conditions is it allowed? When is a test allowed to prompt for information, and what type of information can the test request? Can the user interaction tests be skipped when the full test suite is run in a lab, but enabled when testers run the tests individually?
9. *Support Files and Temporary Files*. If a test case needs to load or read information from a data file, where will the file be located? If the test needs to create a temporary file, where should the test create the file? Should the test be required to delete its temporary files when finished?
10. *Documentation*. How will test cases be documented? How will library routines be documented? What type of documentation should appear as comments in the source file?
11. *Coding Style*. What variable naming conventions should be used (e.g. should variable names for integers start with *i*, variable names for strings start with *s*, etc.)? How and when may global variables be used? What should be in the "main" part of the code, and what should be made into a separate function or subroutine? At what point should code be moved from a script file to a common library file? Should there be a maximum length in terms of run time or lines of code for test cases or script files? Shorter simpler test cases will be easier to write and maintain, but longer complex test cases may expose more bugs.
12. *Script naming conventions*. How should the scripts be named? Should the naming reflect the functionality tested? Should scripts be named based on a number or test ID in a test plan?
13. *Source Management*. Who has permission to modify the source code for individual test scripts or for the library? When updating library files, what review process should be in place to ensure changes are not made that will break test scripts?
14. *Wrapper Functions*. I strongly recommend including wrapper functions for every function native to the test tool as a part of the library. A wrapper function is any function written to implement functionality provided by the test tool. Initially wrapper functions will call the function from the tool directly. Over time, the wrapper functions may be modified to add functionality not natively present in the test tool, or to write a workaround if the function native to the tool ceases to work properly.

Methods to ensure everyone adopts and follows the standards once they're written

Standards are only useful if everyone on the team adheres to them. If standards are not followed, then there is little point in having them. There are a few things that can be done to ensure everyone follows the standards as they're written.

The most important thing any team can do when drafting standards is get everyone involved. People are much more likely to follow the standards if they have an opportunity to give their input, and have an understanding of why certain standards were created. I suggest setting up team meetings to draft standards. At the meetings, discuss all of the items that should be included in the standards, and ensure everyone who has a comment about a standard has the opportunity to be heard. There may be some shy members on your team, and using a round-robin approach, going around the room, asking each individual for their opinion on a topic is a good method to ensure everyone, even the shy folks, have a voice in the process.

The next step to ensure standards are followed is to have them well documented. I recommend creating a standards document based on the meetings where the standards were drafted, and then call additional meetings for everyone to review, and provide comments on the document. It's possible the individual(s) writing the document missed a crucial point. Having follow-up meetings will ensure the document is written properly and is clear to everyone. A well-written standards document is especially important in the long term as personnel on the team changes, and the newer folks need to have an understanding of the standards used by the rest of the team. For most teams, the standards document is a living document, and will require modifications over time. Whenever changes are made, it's equally important to have everyone involved and ensure everyone understands the details of the changes.

Another way to ensure the standards are being followed is to schedule regular code reviews. Code reviews can be done formally, where one tester examines the code of another tester line by line, and does a report on how well the code meets the standards. Another method is to setup a meeting, and distribute printed copies of source code for public comment. A more informal method that may be used is to ask someone other than the script author to triage failures. The individual debugging the script will often be able to provide valuable feedback to the script author.

Conclusion

Developing standards for test automation can be time consuming, but they have several long term benefits, including:

1. *Shared Code.* New library routines are immediately available to everyone on the team.
2. *Efficient Maintenance*. Frequently, maintenance can be done in a single place in a library file rather than several locations in individual test scripts.
3. *Increased Productivity*. New testers will be productive sooner since they only have to learn one set of library routines and coding practices.

4. *Common Environment*. It will be possible for testers to run each other's tests, or to run tests in a lab provided the test machine is configured to the standard environment specifications.
5. *Ease of Debugging*. Standard log file formats will make it easier for testers to debug each other's tests.

Acknowledgements

I thank Noel Nyman, Rom Walton, and Matthew Call for their input and for their feedback on early revisions of this paper.

References

- Hendrickson, Elisabeth, "Making the Right Choice", *STQE Magazine*, March/April 1999
- Hendrickson, Elisabeth, "Evaluating Tools", *STQE Magazine*, January/February 1999
- Powers, Mike, "Styles for Making Test Automation Work", *Testers' Network*, January 1997s
- Links to these, and other articles may be found on the web at Bret Pettichord's *Software Testing Hotlist* at http://www.io.com/~wazmo/qa/

About the Author

Brian Tervo is a Test Lead at Microsoft for the Windows Application Experience Test Team. He was instrumental in bringing his team together to develop standards for their test automation efforts. In Brian's five years of experience, he has worked on three different versions of the Windows Operating System, and has written automated test cases for dozens of retail applications using Visual Test. He has also developed libraries of routines in Visual Test that are being used by the Application Experience team as well as several other test groups at Microsoft. Brian was an attendee at the second Austin Workshop for Test Automation (AWTA2).

Brian Tervo

Brian Tervo is a Test Lead at Microsoft for the Windows Application Experience Test Team. He was instrumental in bringing his team together to develop standards for their test automation efforts. In Brian’s five years of experience, he has worked on three different versions of the Windows Operating System, and has written automated test cases for dozens of retail applications using Visual Test. He has also developed libraries of routines in Visual Test that are being used by the Application Experience team as well as several other test groups at Microsoft. Brian was an attendee at the second Austin Workshop for Test Automation (AWTA2).