Software Test Automation – Developing an Infrastructure Designed for Success

Bill Boehmer

Bea Patterson

Abstract

Automation tools are often viewed as a cure-all that will instantly reduce test cost and effort. However, without up-front planning and infrastructure design, automated tests can quickly become difficult to create and maintain, and the tools nothing more than expensive shelf ware. This paper describes how to initiate a successful automation effort by developing standards and processes for automation and an infrastructure designed for success.

1. Intro

Attendees at any software quality related conference eventually find themselves confronted by an endless barrage of tool vendors trying to hawk their wares. "Test Automation!" they bellow, "The future is now! Test case generation in the blink of an eye and the click of a mouse! Reduce test time and increase productivity! Synergize the dynamics of your process! Stay competitive in tomorrow's world exceeding internal and external customer expectations, with 100% on-time delivery and zero defects! All at the touch of a button!"

What the vendors don't tell you is that 80% of all automation efforts fail. (Then again, neither would I if I were in their shoes.)

Nor do they happen to mention that the process of automating test scripts is expensive and time consuming. The truth is that it takes from three to 10 times as long to create and run an automated test as it takes to run the test once by hand[1]. Three to 10 times! That's quite an investment.

The simple fact of the matter is that test automation is **not** a cure-all that should be taken lightly. It will not solve all of your problems right out of the box, nor will your automation effort be successful without a well-designed plan and a lot of hard work.

Because purchasing and using an automated testing tool is a major investment, we need a means to get a return on this investment. That means? Two means: regression testing and re-use. The benefit in creating automated tests comes from running these automated tests on subsequent builds/releases. This is the ultimate goal of automated testing. **Finding**

[1] Cem Kaner, "Improving the Maintainability of Automated Test Suites". www.kaner.com

bugs is the reason we test, but not the reason we automate – we automate to save time and money, to reduce boredom and make test execution 'easier'. We accomplish this by creating automated tests that can be run on every build of software that needs to tested - today, tomorrow, and for years to come.

This long term cost savings was our initial goal at Siemens Building Technologies, Inc. (SBT). This paper will describe the standards, processes, and infrastructure created at SBT to make automation a success.

2. Background

The first SBT software test automation effort started in 1993. At this time there were only a few tools on the market. After doing some research with the various vendors, we identified two tools that we thought would meet our requirements. We obtained demo copies of both tools and attempted to use them with our applications.

The first tool was difficult to use, making automating extremely difficult, and playback impossible. The tool relied extensively on screen captures; our product at the time was under going numerous changes, forcing us to re-record our automation every time the application changed.

The second tool appeared to be much easier to use and far better suited to our product. But when we attempted to automate several of our test procedures through record and playback we soon discovered the limitations of the tool. We could not record an entire test and get it to playback properly due to many timing and recording issues. A tester with a development background attempted to program around these issues in the scripts. The tool allowed for some programming but was not built-in to the tool in a usable fashion. Scripting was possible only with enormous overhead. This soon proved to be too time consuming and it was next to impossible to generate a usable number of tests.

Around this time, new tools emerged on the market, purporting to be the next generation in test automation tools. The new tools had more robust scripting languages than their predecessors. We evaluated one new tool and found it better suited to our needs than our current tool. Since we had not been using the current tool very long, and had not invested too much money in it, we decided to shelve it and switch to the new tool. With the new tools advanced scripting capabilities we also tried a new approach. We approached test automation with the same process as software development, taking time upfront to defining standards and processes for automation. These standards and processes enabled us to develop a successful automation effort.

3. Our product(s)

Our core product is a Windows based client-server application that monitors and controls the HVAC control systems in large buildings and complexes. The software was designed so that additional options (e.g. fire and security systems) could be integrated into the system as the product family grew. With this in mind we had two main goals for our automation effort. One, we wanted a core set of tests that could be re-run with each release of the product. Two, we wanted automated scripts that could be re-used on future options with as little re-work as possible.

4. Tool selection

We began our automation effort by choosing a tool. The key to selecting a tool is to find the one that works best with your product. There are a lot of advanced tools available -- follow standard decision making practices to determine which is best suited for your use. Some important factors to consider when evaluating tools are:

1) The tool must recognize your product's controls and be able to integrate with your product effectively. There is always the opportunity for you to add 'hooks' into your application later, but you will save a lot of work and headaches if you purchase a tool that can recognize your product right out of the box.
2) The tool must allow you to go beyond simple record and playback. It must allow data-driven test cases and have a programmable language. We'll talk more about this later.
3) The tool must have built-in error logging and recovery. You want a tool that not only logs an error when it occurs, but is also able to return your application to a base state and continue with the next test case. You don't want an entire test suite to stop if the first test case fails. Log files should contain enough information to enable the test executor to determine when the test failed and why.

5. Standards and processes

Once a tool has been selected (and people have been trained on the tool) some standards and processes must be created. Why create standards and processes?

Picture this:

Three people are assigned to the task of automating the testing of an application. Each is given a set of tests to automate. Each creates their automated scripts; all the scripts run; all is well. Then there is a change to the way the software operates (believe it or not). Suddenly, each of the tests needs to be modified! Problem is, test Automator 1 is no longer working on the project. Automator 2 used straight record and playback and must modify almost every script he generated. He barely has enough time to fix his own code, let alone Automator 1's. Automator 3 used some standard coding practices and is able to modify her code easily, but she cannot understand Automator 1's uncommented code. Automator 2 quickly becomes frustrated having to make the same change to every script and soon gets a job elsewhere (it wasn't hard what with all that automation experience now listed on his resume). Automator 3 scraps Automator 1 and 2's work and wants to start from scratch. Management sees that 66.67% of the automation effort has failed and pulls the plug on the entire effort. Back to manual testing! Automation is all hype! Automation does not work!

So how do we avoid such a horrific catastrophe? Simple -- fire all management…

Or, create standards and processes … your choice.

6. Script Standards

Script standards define the format the automated code will take. The standards identify 'standard' methods of documenting features common to all your automated scripts. This way all your scripts will 'look' and 'feel' the same, enabling easier re-use among the various members of the test team. By following simple scripting standards, each script can be constructed so that it is understandable to all, regardless of author. And in the unfortunate circumstance that a valued test-automating employee leaves, his code can now be taken over by a different tester with minimal pain and effort.

We found the following standards to be extremely useful at Siemens Building Technologies, Inc. Most are recognizable as common coding standards, applicable to any type of programming:

6.1 Header files

Header files should be included in every test script. Minimally, they should indicate a list of updates, w/ date and author, and a list of functions/test cases in the file. A brief explanation of the script's purpose is also useful.

For example:

```
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// TWClassLibrary, NewDialogBox Class
// File Name:     newdbx.inc

// Revisions: (newest top,oldest bottom)
// 4     01/15/1996 William Boehmer added error handling in VerifyDBContents.
// 3     11/21/1995 Mel Smith           added new static text to DGGetFields.
// 2     08/15/1995 William Boehmer added field handling to VerifyDBInitialValues
// 1     05/15/1995 Dieter Megert       initial creation.

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Copyright (c) 1995-2001, Siemens Building Technologies, Inc., Buffalo Grove, IL
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

// Comments: This new class contains the following generic test methods to test Dialog
// boxes:

//  1) VerifyDBContents         : Verifies the contents of a dialog box.
//  2) DBEnterFields            : Enters data in all entry fields of the dialog box. If an
//                                error occurs, it will enter all fields, before raising an error.
// 3) DBGetFieldValues          : Gets the data from all entry fields of the dialog box.
// 4) DBClose                   : Closes the dialog box via SysMenu.
// 5) VerifyButtonInvokedWindow      : Verifies clicking on Button opens correct window.
// 6) VerifyDBEverything             : Verifies everything in the DialogBox.
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

6.2 Comments

/* Liberal use of comments is key to maintainable code */

This is extremely important as code will likely be revisited or re-used after it is written.

6.3 Naming Conventions

Having standard file naming convention enables everyone to know what each file is responsible for and what it contains.

For example:

pte_applib.inc = = <u>App</u>lication <u>Lib</u>rary file for PTE application.

Additionally, the various functions and methods used in scripts and in libraries should have common naming conventions. Names should indicate the file the function is located in. For example:

lib_pte_ConvertValueToDMC = = function in pte_applib.inc

local_pte_GetFieldValues = = function local to test script

Standard conventions for variable names are also important in automation, as they are in any programming endeavor (remember, automated testing is programming!)

6.4 Driving Data

A quick look in any test automation book will tell you one thing – automated tests rule!

They will also tell you that automated tests should be DATA DRIVEN!

Straight out of the box, most tools allow the user to record tests and play them back, or what as it is more commonly referred to in the industry: 'record and playback'. This is one of the features that have raised test tools out of the dark ages and into the ease-of-use limelight. However, when used incorrectly record and playback can lead to the untimely death of an automation effort. The paradigm of record and playback is wrought from the noblest of intentions: it allows users to simply open a new test script, record their actions (i.e. the test case), and then play the test case back whenever the testing needs to be run. A few simple steps, and voila -- testing is complete with minimal training and effort … in a perfect world, anyway. In the real world, however, there are several obstacles that make simple record and playback impractical to completely unacceptable. Granted, there are some cases where the application under test or automated tool itself requires extensive use of record and playback, but in most cases it is something to avoid.

There are several problems with record and playback that I will briefly mention in a valiant attempt to make you so aghast at the whole debacle that the mere term 'record and playback' will send shivers up and down your spine every time you over-hear its mention 'round the ol' water cooler, or rather 'Frappuccino machine' (I'm down.):

1) If you want to test (i.e. 'playback') the entry of 100 forms into a database you will have to record (i.e. 'record') the manual entry of 100 forms into the database.

Granted, entering 100 forms one time is better than entering 100 forms every test cycle, but it’s still 99 times too many for me.

2) If the field “User name” changes to “Name” you will have to search the scripts and change all 100 instances of “User name”. Multiply this times the number of times marketing decides to change the names of fields throughout the project (not to mention the name of the project itself) and one quickly sees the horrific cut and paste nightmare that awaits.
3) If the field “User name” is moved to a separate dialog box, all 100 form-entry tests must be changed. Multiply this times the number of times programmers decided to change their minds, and one quickly finds a new profession.
4) If the tester recorded something incorrectly or decides something should have been recorded differently, the 100 tests must either be updated or the entire script re-recorded.
5) Tests relying on record and playback tend to “code in” errors. If a bug exists where an erroneous error message appears, the recorded test EXPECTS this dialog box and passes, even though a bug exists in the software. This is never good. Ask anyone.

Given these examples, it is eminently obvious that record and playback is not the solution.

But before you hastily scrawl (or rather, aggressively type) a letter off to your tool vendor of choice lambasting their proffering of such a worthless feature as record and playback, I should point out that record and playback **is** a wonderful tool. The solution, as you might have gathered from the subtly placed capitalization at the start of this section, is to adopt a **data-driven** methodology.

What does this mean? To put it simply – record a test case then variablize the data. In the previous example, the entry of 1 form should be recorded once and then modified to allow 100 forms to be entered based on the single recording. If you replace the recorded data with variables, you can create a generic test case where the ‘data’ is passed (or ‘driven’) into the test case. In the previous example, this would mean changing the data recorded, e.g. the User name “Joe”, to a variable, e.g. sUsername.

The line:

```
Form1.Username.Enter (“Joe”)
```

Then becomes:

```
Form1.Username.Enter (sUsername)
```

And sUsername becomes “Joe”.

A list can then be created containing all 100 of the sUsername’s, and each passed into the test case, one at a time.

By variablizing all of the data, we then have an array of 100 lists of data to be entered (all in one place for easy modification), and one generic test case (for even easier modification.)

Simple, no?

Automators eventually learn to use a combination of record and playback and straight scripting to crank out perfectly scripted data-driven tests.

6.5 File locations

Create a standard file structure on the test PC. By doing this, the file structure can be created on any PC, and the test scripts can be run on any PC. A standard file structure also makes it easy for any automator to find any file on any PC.

7. Test Case Standards

One of the most important standards, relating to testing in general, is that test cases should always be written to be independent of one another. As stated earlier, you don't want an entire suite to stop execution because the first test case failed. Believe me when I say this is not something you want greeting you first thing Monday morning when you walk into the office (or so I hear.)

To avoid this Monday morning wake-up call from hell, your test case standards should clearly state that test cases should be independent of one another. While this is not always possible, it certainly ought to be the goal (just because something is not possible 100% of the time does not mean it should not be the ideal).

A simple structure that promotes test case independence is as follows:

- Pre-amble
- Test steps
- Post-amble

The pre-amble contains the setup conditions for the test case (variable declarations, reading in data from a file, getting the application to a base state, etc.)

The test steps are the steps one takes to execute the test. Also included should be mechanisms to log results to the results file.

The post-amble contains the exit conditions for the test case. Typically this involves cleaning the application up after the test case, and returning it to base state.

Bonus hint!

Rather than fail the test case where an error occurs in the test steps, log the error, putting enough information into the result file to aid error analysis. Then fail the test case in the post-amble, after the application has been returned to base state. This helps to identify where the test failed, and allows an iterative test to continue after the first error. For example, if you have a test case where five fields in a dialog box are being checked for information, rather than fail on the first field and skip the rest, you want the test case to log the error on the first test case and continue with the other four fields. Fun and useful!

8. Infrastructure

Standards will organize how your testers will automate, ensuring they automated test scripts are coded consistently and uniformly.

Once you have created standards, you need to design and develop an infrastructure that will support your automation effort. Creating an infrastructure is essential for code re-use because it helps you create libraries of functions that can be used by any automator with any test script.

8.1 Automation Library Infrastructure

The automation library infrastructure is the code base the individual test scripts are built on. It is literally the foundation of the automation effort. The infrastructure contains all the common libraries used in the automation. The libraries contain functions that are common to individual applications, multiple applications, or completely generic to the application under test (e.g. file manipulation routines).

These common functions can be thought of as the “business layer” for the automation. By designing an infrastructure in this way, the test scripts themselves then become the “display layer”, containing the data to be entered/verified and simple calls to the business layer. The scripts present the data being tested, and the libraries contain the test logic. In this way, you can create a solid foundation of reusable code logic that can be used by any test script.

To get the most out of test automation, it is important to have the infrastructure in place early and the process for populating the libraries defined so that the entire team can make use of them and contribute to their population. At first, the libraries will only contain rudimentary functions. Then, as the automation effort proceeds, more advanced functions from individual scripts can be added to the infrastructure library. When a function is found being used in more than one script, or can be used across multiple scripts, it should be added to the library. By populating the library, you create functions only once and call them from the individual scripts, rather creating the same functions over and over again in each script. This allows teams to solve problems once and be robust, rather than re-invent the wheel each time. This library structure allows for the re-use of existing automation code to avoid redundancy and accelerate test script creation. Additionally, by having code that has been run by numerous scripts, bugs in the test code will be identified and fixed quickly, resulting in more stable automation.

By following the scripting standards you have created, your testers will create library functions that all the automators on the team will be able to understand and be able to use.

Libraries created for one project can then be re-used in others, often with little or no modification.

To aid usage, we created three types of libraries in our automation infrastructure: Class, Tool, and Application specific.

8.2 Class Library

The automation tool we used at SBT allowed us to create new window classes based on standard window classes. New functions were created specific to these new windows classes. Objects in the applications belonging to the standard window classes were re-assigned to these new classes, thus giving them access to these new functions. By associating standard functions to the window class rather than the specific instance of the class, generic test functions could now be used by any test script testing any application of the product.

Class Library files contain all the new classes and their functions. Since all the functions are generic, the Class Library is independent of any application or project.

For Example:

A new class NewDialogBox was created based on the standard DialogBox class.

A new NewDialogBox class function was created named DBEnterFields:

void DBEnterFields (WINDOW wDialogBox, LIST OF WINDOW lwFields, LIST OF ANYTYPE laData, BOOLEAN bOP optional, WINDOW wAccept optional, BOOLEAN bRaiseError optional)

All instances of DialogBox class were re-assigned to NewDialogBox class so they could use the new DBEnterFields function.

New functions specific to the DialogBox class windows are now added to the NewDialogBox class library, giving all instances of NewDialogBox's access to the new common functions.

8.3 Tool Library

The Tool library files are populated with generic functions that can be used by any test script testing any application or product. (These are typically file or string manipulation functions, or any function existing independent of the application under test).

For example:

STRING lib_ENV_MakeXDecimalPlaces (STRING sValue, INTEGER iNumOfPlaces, BOOLEAN bPad optional)

8.4 Application Specific

The Application Specific library files (AppLib) contain functions specific to certain application windows. Each application being tested has its own AppLib file.

For Example:

ANYTYPE lib_pte_ConvertEnteredValueToExp (LIST OF ANYTYPE laValues, WINDOW wDialogBox, LIST OF WINDOW lwListofFields, WINDOW wField)

9. Process

Once you've developed standards, you'll need to develop a process for automation. The process you define should cover how to select automation targets, how to manage code change, and how to handle the interaction of different automators working with the same code base.

9.1 Who should automate

It is important to realize that automated testing **is** programming. The standards and processes followed are those of a software development process. When choosing people to write automation code it is important to hire people who have some type of coding background, or to train your automators in programming concepts. Given the programmable syntax of most test tools, and the fact that proper coding methodologies are essential to getting the most from your test tool, it is easy to see where programming skills are a necessity for a successful automation effort. Automators must be able to understand and apply the various standards and processes we have created for automation, as well as understand the importance of design before coding, to get the most out of maintainability and re-use.

9.2 What to automate

Even with well-defined standards and an impressive infrastructure, you can't just set a team out to automate tests without direction. You first must define a process you will use to determine what will be automated.

Use the following guidelines when setting up a criterion for automation targets:

1) Automate regression tests: Since the return on investment in automation comes from script re-use, you should first target your regression tests. Automate any tests that will have to be run with every build (e.g. Build acceptance tests, installation tests, or other high priority tests). Only automate tests that will be repeated. One-time tests are not worth automating.
2) Automate tests for stable applications: Before automating the testing of an application, look at the application itself. There is no point in beginning automation on an application that is likely to change in the future. If the application changes, the automated tests must also change, and re-work is never good (or fun). Therefore, only automate tests for applications that are stable.
3) Automate non-time dependent tests: Do not automate tests with complex timing issues. Many times able bodied automators feel the need to code their way out of complex situations. While this may be fun, it is also a big waste of time. If a test is too hard to automate run the test manually. Automation is not always the answer, and 100% automated testing should not be the goal. You want to increase productivity, not waste a week on the automation of a test that takes 5 minutes to run manually, so leave the overly complex tests for manual testing.

4) Automate repetitive tests: If a test is extremely repetitive and boring, please, please, please automate it. The sanity you save alone will be worth the investment. Test zombies never prosper.
5) Automate tests that have been written: You must first have detailed test cases that are repeatable before you start automating. Always write test cases before automating them. This ensures that tests are written in terms of application functionality, independent of the automation effort. **If tests are written by automating the automation becomes the focus rather than the testing.**
6) Limit your scope: Don't try to automate everything. Achieve small successes then increase your scope as you make progress. Most software development teams are all constantly in and out of schedule crunches. Make the most of the time you have, and grow your automation effort when you can.

9.3 Configuration Management

A configuration management tool should be used to store scripts. Using a tool enforces test code change management by keeping a record of all changes made to scripts. It also allows you to rollback to previous versions if necessary. This is useful not only in maintaining control of test script code, but also in the event that past versions of tests need to be re-executed.

Using a configuration management tool is also useful when several automators are working on the same code. These tools control script management by only allowing one person to update a file at a time. With this type of code sharing it is also important to have a process for updating common files. This process should include some ground rules to keep individuals from stepping on one another's work. It is important in these situations that no change is made to any common library file that will cause scripts that use functions in the library to fail. Modification should be done in such a way that existing scripts are able to use the functions without modification. You can accomplish this in most automated tools by adding optional variables to the function declaration. Each tool is different, but this should always be the goal. To enforce this, it might be beneficial to setup a process for reviewing library changes so that people don't make modifications to the library that will break others scripts. This process can be as simple as stating every change must be reviewed by at least two other automators, or having a lead automator responsible for reviewing all changes.

9.4 Start to automate and build libraries

Once the standards, processes, infrastructure, automation targets and criteria for selection have been created, automation may begin!

This is when everything comes together and you have a team of efficient automators following standards and processes to create uniform code, actively populating libraries, reusing one another's code, and automating those features and functions specifically targeted for automation.

10. Automation Successes at SBT

Automation on our HVAC front-end applications at SBT has been going on using our current tool since 1995.

To date, eight application tests have been automated (100's of test scripts). These scripts are re-run on every test cycle, and are intermittently run to verify regression functionality. Additional test scripts are added on to existing suites for every product release.

The scripts have helped reduce overall test execution time. One application test time went from 55 hours to 6 hours. Another's was reduced from 21 hours to 7 hours.

The automation libraries created for our base product have been re-used on two subsequent projects. For these projects, the library files were used as is and new test scripts created. The test scripts simply read in data from a .csv file and pass the data to several test cases. The test cases are made up of simple calls to library functions and some logic for special case handling. The time spent to automate the testing on these projects was minimal. The creation of the test scripts in both cases took less that one week. The biggest effort was in the creation of the data files containing the information to input into the application. On one of the projects an experienced test automator created the test scripts, and a tester with no automation experience but a large amount of product and database knowledge created the data files. This allowed each tester to work in those areas they were most knowledgeable in and could be the most effective.

11. Endtro

Test Automation is not a cure all that should be stepped into lightly. Starting an automation effort is a large investment that requires upfront work and design in order to be successful. By creating test automation standards, you can ensure tests will be created in a way such that they will be easy to use and maintain. By setting up automated library infrastructures you can create a system in which test code can be re-used between testers, applications, and in some cases, projects. By defining processes for use, you can ensure code changes are managed, and that the scripting takes place only in those areas that are well suited for automation.

Together these standards and processes have enabled us to create maintainable, re-useable software test automation at Siemens Building Technolgies that continues to grow each year.

Bill Boehmer

Bill Boehmer has worked for Siemens Building Technologies for over 6 years. Bill is a member of the software test process improvement core team at SBT, and is considered one of the experts in test automation and has assisted many teams within the organization in initiating automation efforts. He has contributed to many of the test standards that are used within the company for software testing. Bill is currently the test lead for two new development efforts within the organization. Bill has a B.S. in EE from Marquette University.

Bea Patterson

Bea Patterson has over 15 years experience in both software development and software testing. She has worked for Siemens Building Technologies for over 15 years. Bea has been involved in all areas of software process improvement and software testing. She is the leader for the software test core team in the organization and her major responsibilities include leading the core team, developing test standards, and identifying areas of test process improvement. Bea is currently the test lead for one of the new development efforts within the organization. Bea has a B.S. in CIS from DeVry Institute of Technology.

Recently Bill and Bea presented a paper on "Cultivating an Organization's Test Discipline" at the 1999 Practical Software Quality & Testing Techniques North Conference.

Contact Info

Bill Boehmer – bill.boehmer@sbt.siemens.com
Bea Patterson – bea.patterson@sbt.siemens.com

Siemens Building Technologies, Inc.
1000 Deerfield Parkway
Buffalo Grove, IL 60689