Getting Started With Automated Testing

Michael Kelly
Mike@MichaelDKelly.com

Bio: I am a software testing consultant for Computer Horizons Corporation with experience in software development and automated testing. I have published numerous articles on topics in test automation and testing in general and have been a technical editor for IBM Rational Software for articles on various testing topics. I am a former adjunct professor in computer science at Indiana Institute of Technology where I taught a course in software testing.

Abstract: This paper outlines the key steps to take as you get started with automated testing. These ideas should be especially useful to you if you are doing automated testing for the first time. This paper is not geared to a specific test tool or tool vendor, and will offer sound advice if you have already selected your tool, if you are looking to purchase a tool, or if you are building your own tools. This paper does not address the day-to-day details of test automation but focuses on developing a strategy for automation, putting together an automation team, and some simple first steps.

Welcome to Automated Testing!

This paper outlines the key steps to take as you get started with automated testing and should be especially useful to you if you're doing automated testing for the first time ever. This paper is not geared to a specific test tool or tool vendor, and offers sound advice if you have already selected your tool, if you are looking to purchase a tool, or if you are building your own tools. This paper does not get into the day-to-day details of test automation but instead, consists of a series of simple steps that help you look at automation at a high level, focusing on developing a strategy for automation and putting together an automation team.

Throughout the paper, I refer to building the right automation team. Just about every team will benefit from *Set Goals for Automation* and *Set Some Standards*. Depending on your context, some of the may be more valid or necessary than others. For example, *Document Everything* may be the most important step you take if you work in an FDA regulated environment, but documenting everything may not be as practical if you are working in an XP environment. There is no particular order in which the steps need to be executed.

This article focuses on team makeup and coordination. Some of the steps will only be applicable for teams of more than one person. Some of the examples I share in the paper are from when I was working as the only automated tester for a project while others are from projects where there were up to eight full-time automated testers (not to mention the other test engineers). The steps discussed in this paper apply primarily to teams of three or more people. If your automation effort has only one or two full-time automated testers there is still good information available to you here, though not all of it will apply.

Terminology

Automated testing is the use of tools assist with a testing effort. There are many types of test automation, including automated unit testing, functional regression testing, and performance testing. Test automation can be as simple as automating the creation of some of your data, or as complex as a series of scripts/programs that do parallel comparative testing of two or more systems with live data. A small sampling of different types of automation available is listed below:

- Load and performance testing
- Installation and configuration testing
- Testing for race conditions
- Endurance testing
- Helping the development effort with smoke tests and unit tests
- Analyzing code coverage and runtime analysis
- Automation of test input generation
- Checking for coding standards and compliance
- Regression testing
- many more…

The main focus of this paper will be on scripted automated testing (be it performance testing, regression testing, unit testing, or some other such type of scripted testing). Some of these steps will be applicable to all of the types of automation listed in the bullet points above (for example *Set Goals for Automation*) while others (such as *Look for Ways to Modularize Your Scripts*) will only apply to scripted development.

Automated testing is a subset of the overall testing profession. There is an overlap of the skill set for those who make good testers and those who make good automated testers. In a couple of places I will talk about skills needed specifically for automated testing, but one should keep in mind that an automated tester should be a tester first. That is, they should be skilled in the arts of black box testing, white box testing, domain testing, performance testing, exploratory testing, and any of the hundreds of other types of testing.

Working with these two suppositions (there are many ways to perform automated testing and automated testers should be testers first) we can now jump into some of the key steps that will make your automated test team successful. First, we will look at setting some goals for test automation. Setting goals can add focus and a sense of mission to the automation effort. It can also ease the pain of determining your return on investment, as unlike functional regression testing, many of the types of automated testing have no easily quantifiable return.

The first step when getting started with automation is to ensure that you are setting the correct scope for your effort and that your efforts are focused on the right goals. In this next section, we will look at goals for automation and how they will affect the rest of your decisions.

Set Goals for Automation

Again, there are many different types of test automation. One of the problems most teams encounter is that they only focus on a few of the more popular types of automated testing (functional regression testing, unit testing, performance testing, etc…). Setting goals for automation can affect what you automate:

- Do you want to find bugs?
- Do you want to establish traceability for some sort of compliance?
- Do you want to support the development team?
- Do you want to establish scripts that allow you to ensure nothing changes in the software?

These questions are a small sampling of all of the questions that need to be asked in order to determine the scope of your automated testing. Related to your goals, your software development lifecycle and the skills of your testing staff will also affect what you choose to automate. Are you developing in an XP lifecycle with programmers also testing? This would probably lead to more automation[1]. If all of your testers are junior, you probably will use very little automated testing or automated tests that are shallow.

Use your goals to set your strategy and scope for your automation. If your goal is to support your developers, select tools and training to support that goal. You will need testers who know how to program and who can communicate with developers in their language. They will need to look at source code, configure environments, and will probably spend a good deal of time working side-by-side with developers debugging and troubleshooting. In contrast, if your main goal is to support refactoring and change, you will have a team with experience developing regression scripts and need to have team members with both strong business knowledge and strong testing skills. They will need to know how to create scripts that look for changes and will need to know the business in order create scripts that look for underlying changes that may be taking place.

A common mistake when automating for the first time is biting off more than you can chew and consequently missing deadlines or having to work unrealistic hours in order to meet them. Either of these situations will demotivate your testing team and make them look bad in the eyes of the rest of the development team. By setting your goals, you can start small and keep things simple for your first-time automating. In future iterations or projects, you may want to go crazy and automate everything, but by starting small now, you will minimize possible rework later when you start adding new tools or new automation techniques.

When deciding what to automate for your first time, start with small milestones. For example:

- If you're testing a GUI or Web application, start with testing simple functionality. This could include verifying that all the correct controls exist on the screen, the proper fields enable/disable when actions are taken, etc….

[1] In past projects I've worked on, when the test team worked closely with the development team, we had much more automation and the resulting scripts and/or environments were more stable and more robust. On projects where we sat in different buildings or a significant distance apart from them, the test code was less effective and we had more intermittent test bugs due to code workarounds.

- If you're automating performance testing, start with just one virtual user and set your goal at a low number (no more than twenty). When you get one virtual user to work, double this and get two to work. Keep doubling this until you get to twenty. Each increment could present a new set of challenges.

Whatever you choose to test, make sure that it doesn't span more than one part of the application-under-test or more than one or two Web pages. Ideally, you should be able to use record-and-playback features to perform this basic testing or you can do some basic scripting (hand-coding scripting commands into the script files instead of using record-and-playback). These basic scripts will then become the baseline moving forward. Many testing tools will have advanced features such as the ability to add delays and timers, the ability to distribute testing on different machines, and the ability to create graphs for just about everything. Stay away from these as much as possible at first, because they'll just confuse you. Only after your team has had some small successes should you explore these useful and often necessary features.

Once you have determined your goals, you will need to start thinking about what kinds of skills you will need to have on your team. The next step addresses one of the most common oversights when staffing an automated testing team. In your context, depending on your goals, you may need more development-oriented automated testers, or you may need some other special skill.

Have at Least One Programmer on Your Team
The most important thing you can do is select the right team. I recently saw a company do a comprehensive analysis of three market-leading enterprise level test tools. They compared them side-by-side in different scenarios, using their most skilled staff to run the tools through their paces. A couple of months later the team came back and said, “this tool is the tool for our company.” They gave little to no thought about who would be using the tool, their technical experience, where they could get training and how much it would cost, if they had past experience with one of the other tools, etc… This is the biggest mistake I think you can make. Your human resources are your biggest investment and they will be the determining factor for your automation success. A tool is and always will be, only a tool.

To ensure efficient well-planned automated testing, you will need to have at least one experienced programmer in your testing-automation group. You'll soon find out that automated testing *is* code development. If you will be doing a lot of scripting in your automation effort, most likely you will want to build modularity into your scripts. You will want to create some custom functions, extend the test tools, and potentially read test data from databases or data files. Depending on the tool and language, encapsulation, object orientation, or some other method may be used to make the code more maintainable or easier to use. If you are looking at providing code analysis services (performance profiling, code coverage, runtime analysis), you will need someone who knows how to read the code-under-test and who can ask the development team the right questions.

Even if the scripting environment you are using is not Java or C++ (which it may very well be), you're still building systems of scripts, data files, and libraries. Record-and-playback features only offer quick solutions for the most common tasks and controls. For advanced automation of any kind or for any custom controls, you'll need to be able to write your own *maintainable* code. That means employing programmers, not manual testers who learn to code as they go. Having said that, beware you don’t employ only programmers. All team members should be testers first. Skilled in the mental arts of testing as well as armed with the ability to code and design test systems.

That said, once a decision has been made as to what tool will be used, it is important that the users of that tool know how to use it properly. The next step offers some places to look if you are still choosing your tool, and offers some advice on ensuring the automation team knows how to use them.

Get Familiar with the Tools
Depending on what tools you already have in your arsenal and what types of automated testing you will be doing, this section may or may not apply to you. Most automated testing tools are designed for unit testing, some sort of regression testing, or performance testing. Addressing what tools are available and what tools may be best for you are outside of the scope of this paper. However, if you are still reviewing

tools and would like some help selecting the right tools for your specific context, I would recommend using the following resources (which I am in no way affiliated with):

Open Testware Reviews: http://tejasconsulting.com/open-testware/
- A source of information about freely available test tools.

Testing Tool Information: http://www.grove.co.uk/Tool_Information/Choosing_Tools.html
- Short Sharp Advice about how to choose a testing tool.

Test Tool Evaluation Center: http://www.testtoolevaluation.com/index.asp
- An online resource with a lot of info and wizards for tool comparisons.

The scope of your automation effort will depend heavily on the tools you use and have access to. If you already have an investment in an enterprise test tool, you will probably want to leverage it. If you are operating on a tight budget, you will look more at open source tools, shareware, and custom tool development and scripting (virtually *any* scripting language can be it's own full-blown automation tool). Below is a sampling of some of the different names commonly used to label test automation tools:

- Disk imaging tools
- File scanners
- Macro tools
- Memory monitors
- Environmental debuggers
- Requirements verifiers
- Test procedure generators
- Syntax checkers/debuggers
- Runtime error catchers
- Source code testing tools
- Environment testing tools
- Static and dynamic analyzers
- Unit test tools
- Code coverage tools
- Test data generators
- File comparison utilities
- Simulation tools
- Load/Performance testing tools
- Network testing tools
- Test management tools
- GUI testing tools

Regardless of what tool(s) you select, you will need to be sure that you spend time learning how to use them properly. Go through the tutorials that are provided and read through whatever documentation is available. While the tutorials aren't the definitive guides as far as training is concerned, they do get you familiar with the software as well as any vendor specific terminology. Both of these are important. As you'll soon find out most tools are large and complex with feature upon feature. The tutorials will familiarize you with the features you'll be using most.

If you feel you still need more familiarity with the tools after you've completed the tutorials, attend a training class (if they are available), or hire a training consultant to come in and spend some time with you. Having a basic understanding of the tools is essential. Make sure your whole team has had some form of training or some reasonable amount of time playing with the tools before you try to do any real work. (I find that programmers tend to pick up test tools very quickly, while nonprogrammers struggle with some of the programming concepts and need more time.)

Once you know which tools you will be using and have a good idea of how you will be using them, you should start to think about setting some guidelines for how your team will use them together. The next step looks at setting up standards for tool use. These standards will make training easier, make it easier to move team members from project to project, and reduce the cost of ownership for the tools you choose.

Set Some Standards
You'll also find it useful to develop standards for your automated-testing team. This is just as important in testing as it is in conventional software development. Your test system will develop more rapidly and will be easier to maintain if you establish and enforce naming standards, coding standards, environmental standards, and procedures for error and defect tracking. Having these standards documented will also allow people new to the project team to come up to speed faster.

- **Naming standards** for scripts, test logs, directory structures, data structures, and verification points help to keep everyone on the same page. On two of my last three projects, we maintained more than 1,000 scripts and 5,000 data structures for each project. A good naming standard was the only thing that made that possible.

- **Coding standards** should also be developed and enforced. For most companies this is easy — you can just steal the standards used by the development staff. If you don't have this advantage, go online and find some. They're out there, and you can find a reasonable set of standards in about 15 minutes. Once you've used them for a while, you can customize them to fit your needs and the needs of your team.

- **Environmental standards** should ensure that the computers you use all have the same operating system, RAM, hard drive space, and installed software configurations. The only differences should be the specific differences you are looking for in your configuration testing (if applicable). I've found that many of my hard-to-find and expensive-to-fix bugs, for instance, have been due to the fact that a script was developed on a computer that had more resources than the one on which it was executed.

- **Procedures for error and defect tracking** should describe how to log errors in test scripts, submit defects via your defect-tracking tool, code workarounds into scripts, and remove it all after a bug is resolved[2].

Document your team's standards, and be sure your team knows the standards and follows them. This step will also prepare you for the remaining steps as it will potentially provide a framework for decision-making and review. Next is a step focused more on script-based automation, but the principle can still be applied to other types of automation.

Establish Some Baselines
Once you've decided what you're testing, you should establish some simple baselines. As mentioned above this can be done using record-and-playback features, by performing some simple scripting, by gathering small sets of runtime data, etc…. Whatever type of testing you're implementing, figure out what the bare minimum tests are for that type of testing to be successful and implement them in very small, very simple chunks. Remember that this is your first project and most likely your first time using these tools. You will not have sophisticated frameworks and test architectures in place yet. Those will come later as your team matures. You will use these baselines as the foundation for what you will be implementing going forward. Hopefully, you will have some sort of archiving or source control and can always come back to these simple tests if you need to.

In past projects I have established the following baselines:

- For regression testing: a baseline set of scripts that test basic functionality in the system using the most commonly used paths through the application.

- For performance testing: a baseline set of scripts that target each standalone service for the application that run successfully with one virtual user.

- For code coverage analysis: try to develop a series of tests that will allow you to get some level of coverage in each major module of your application.

- For source code checking tools: try to get a small subset of the application code to pass the verifications and modify/refine the rules as needed.

[2] I didn't figure this out until a few projects ago. My team had many problems communicating when it came to finding and reporting bugs. One of us would log a bug and develop a workaround in the script without communicating to the rest of the team what we had done. Inevitably, someone else would test the bug when it was fixed, mark it resolved, and never remove the workaround in the code. Usually this lack of communication caused confusion and rework, and cost the team time.

- For establishing traceability: take a small series of test cases and, using your test management tool, establish traceability from the manual/automated scripts, to the test cases, to the requirements. Generate a simple report that shows this traceability in a usable format.

- For test data generation: generate a subset of the overall data you will need for testing and verify the data's correctness and randomness.

Once you have established your baselines you will be better prepared to start looking at more advanced methods of scripting (or data gathering, or scenarios, etc…). This is important as the next two steps focus on optimization of these baselines. As you mature in your processes and experience, you will be able to cut out this step, but be sure you are taking small steps right now.

Look for Ways to Modularize Your Scripts
Now that you have your baseline scripts, grab a good software architect - or your team of testers and a large whiteboard. Start looking through the code in the scripts for repetitive calls or other common actions. What you're doing is looking for ways you can modularize your scripts.

Ideally, you want to optimize your scripts so that maintenance is as easy as possible. I've found that I've *never* regretted spending too much time developing a powerful and robust script, and I've often kicked myself for taking shortcuts in development. A script *will* cost you more to maintain than it will to create, unless you develop the script just as you would the software it's testing. Do it right the first time and reap the rewards in all of the following iterations of the project. After you've planned out what modularization you can do, implement it using whatever method your tools allow for the most reuse (libraries, objects, classes, etc…). More than likely, you'll carry these over to following projects, and they'll evolve and change as you do.

For ideas on how to modularize your scripts, look at user communities for the tools you use, read literature on the topic from any of the major testing websites, talk with your development team, or hire in a consultant to train your staff on what to look for. These resources are also good places to find information about the next step as well.

Use Data Structures and Data-Drive Techniques
Effective and cost-efficient automated testing is data-driven. Keith Zambelich's whitepaper *Totally Data-Driven Automated Testing* (available at www.sqa-test.com) is a must-read for anyone doing automated testing. Data-driven testing simply means that your test cases and test scripts are built around the data that will be entered into the application-under-test at runtime. That data is stored by some method and can be accessed by some key used in your scripts.

This method offers the greatest flexibility when it comes to developing workarounds for bugs and performing maintenance, and it allows for the fastest development of large sets of test cases. Most likely, your tool will have some method for implementing this, or you can use a spreadsheet or a database. As a special bonus, once you get good at data-driven techniques, you can use automated methods to generate some of that test data for you.

Document Everything (that makes sense to document…)
Finally, document why you designed things the way you did. Document what each module does, and what each function in it does. All of this documentation is useful as training material or for future reference, and it helps you keep track of lessons learned. I document as much as time allows. Sometimes I get it all, and sometimes I can't document anything. I've never regretted documenting any information, but on more than a few occasions I *have* regretted not having any clue about how something worked or why I made a certain decision on a project I'd been away from for a couple of months. Sometimes documentation is the only thing that can save project scripts that no one has worked on in a while.

A Review
Even if you follow all the steps above, you'll still struggle the first time you attempt automated testing. Just remember to follow this road map and you should survive:

- Set goals for automation.
- Have at least one experienced programmer in your testing-automation group.
- Get familiar with your tools.
- Develop standards for your team.
- Establish some baselines.
- Modularize and build reusability and maintainability into your scripts.
- Use data-driven testing techniques whenever possible.
- Document what makes sense to document.

The clarity and simplicity of your goals, your understanding of the tools, and the makeup of your team. These are the factors that will determine your success.

As your knowledge of these grows, your goals and priorities for testing will develop. The way you automate your testing will also change as you include more tools, get more experience with them, and read about more complex and innovative ways of using them. Primarily, the quality and success of your testing will grow as the skill level and experience of your team changes. Tools will do nothing if people can't use them or don't know how to test to begin with. It has been my experience that the makeup of the team is the most influential factor in an automation efforts success.

Your goals should be driven by your business knowledge. Your programming knowledge and testing skill reflect your team and how they use their tools. Test automation requires a balance between programming knowledge, testing skill, and business knowledge. Incorporating all of those assets into your team will be the most important step of all.